AIFlow

Release 0.4.dev0

flink-extended

Oct 09, 2022

CONTENTS

CHAPTER

ONE

GET STARTED

1.1 What's AIFlow

1.1.1 Introduction

AIFlow is an event-based workflow orchestration platform that allows users to programmatically author and schedule workflows with a mixture of streaming and batch tasks.

Most existing workflow orchestration platforms (e.g. Apache AirFlow, KubeFlow) schedule task executions based on the status changes of upstream task executions. While this approach works well for batch tasks that are guaranteed to end, it does not work well for streaming tasks which might run for an infinite amount of time without status changes. AIFlow is proposed to facilitate the orchestration of workflows involving streaming tasks.

For example, users might want to run a Flink streaming job continuously to assemable training data, and start a machine learning training job everytime the Flink job has processed all upstream data for the past hour. In order to schedule this workflow using non-event-based workflow orchestration platform, users need to schedule the training job periodically based on wallclock time. If there is traffic spike or upstream job failure, then the Flink job might not have processed the expected amount of upstream data by the time the TensorFlow job starts. The upstream job should either keep waiting, or fail fast, or process partial data, none of which is ideal. In comparison, AIFlow provides APIs for the Flink job to emit an event every time its event-based watermark increments by an hour, which triggers the execution of user-specified training job, without suffering the issues described above.

1.1.2 Features

1. **Event-driven**: AIFlow schedule workflow and jobs based on events. This is more efficient than status-driven scheduling and be able to schedule the workflows that contain stream jobs.
2. **Extensible**: Users can easily define their own operators and executors to submit various types of tasks to different platforms.
3. **Exactly-once**: AIFlow provides an event processing mechanism with exactly-once semantics, which means that your tasks will never be missed or repeated even if a failover occurs.

1.2 Quickstart

1.2.1 Running AIFlow locally

This section will show you how to install and start AIFlow on your local workstation.

Installing AIFlow

Please make sure that you have installed AIFlow refer to *installation guide*.

Starting AIFlow

Starting Notification Server

AIFlow depends on notification service as an event dispatcher. Before running AIFlow, you need to start notification server.

```
# Notification service needs a home directory. `~/notification_service` is the␣
↪default,
# but you can put it somewhere else if you prefer.
export NOTIFICATION_HOME=~/notification_service

# Initialize configuration
notification config init

# Initialize database and tables
notification db init

# Start notification server as a daemon
notification server start -d
```

Starting AIFlow Server

```
# AIFlow needs a home directory. `~/aiflow` is the default,
# but you can put it somewhere else if you prefer.
export AIFLOW_HOME=~/aiflow

# Initialize configuration
aiflow config init

# Initialize database and tables
aiflow db init

# Start AIFlow server as a daemon
aiflow server start -d
```

Note: You may run into issues caused by different operating systems or versions, please refer to Troubleshooting section to get solutions.

Running a Workflow

Defining a Workflow

Below is a typically event-driven workflow. The workflow contains 4 tasks, task3 is started once both task1 and task2 finished, then task3 will send a custom event which would trigger task4 to start running.

```
import time

from ai_flow.model.action import TaskAction
from ai_flow.notification.notification_client import AIFlowNotificationClient
from ai_flow.operators.bash import BashOperator
from ai_flow.operators.python import PythonOperator
from ai_flow.model.workflow import Workflow

EVENT_KEY = "key"


def func():
    time.sleep(5)
    notification_client = AIFlowNotificationClient("localhost:50052")
    print(f"Sending event with key: {EVENT_KEY}")
    notification_client.send_event(key=EVENT_KEY,
                                   value='This is a custom message.')


with Workflow(name='quickstart_workflow') as w1:
    task1 = BashOperator(name='task1', bash_command='echo I am the 1st task.')
    task2 = BashOperator(name='task2', bash_command='echo I am the 2nd task.')
    task3 = PythonOperator(name='task3', python_callable=func)
    task4 = BashOperator(name='task4', bash_command='echo I am the 4th task.')

    task3.start_after([task1, task2])

    task4.action_on_event_received(action=TaskAction.START, event_key=EVENT_KEY)
```

You can save the above workflow as a python file on your workstation and remember the file path as ${path_of_the_workflow_file}.

Uploading the Workflow

Now you can upload the workflow with the path of the file you just saved.

```
aiflow workflow upload ${path_of_the_workflow_file}
```

You can view the workflow you uploaded by the following command:

```
aiflow workflow list --namespace default
```

Starting an Execution

The workflow you uploaded can be executed as an instance which is called execution. You can start a new execution by the following command:

```
aiflow workflow-execution start quickstart_workflow --namespace default
```

Viewing the Results

You can view the workflow execution you just started by the following command:

```
aiflow workflow-execution list quickstart_workflow --namespace default
```

The result shows `id`, `status` and other information of the workflow execution. If it is the first time you execute a workflow, the id of the workflow execution should be `1`, so you can then list tasks of workflow execution with id `1` by the following command:

```
aiflow task-execution list 1
```

Also you can check the log under `${AIFLOW_HOME}/logs` to view the outputs of tasks.

Stopping AIFlow

Stopping AIFlow Server

```
# Stop AIFlow server, it may take a few seconds to wait for the server stopped.
aiflow server stop
```

Stopping Notification Server

```
# Stop Notification server
notification server stop
```

What's Next?

For more details about how to write your own workflow, please refer to the *tutorial* and *concepts* document.

1.2.2 Running AIFlow in Docker

This section will show you how to start AIFlow in docker container if you are tired of managing the python environment and dependencies.

Pulling Docker Image

Run following command to pull latest AIFlow docker image.

```
docker pull flinkaiflow/flink-ai-flow-dev:latest
```

Running Docker Container

Run following command to enter the docker container in interactive mode.

```
docker run -it flinkaiflow/flink-ai-flow-dev:latest /bin/bash
```

Starting AIFlow

Starting Notification Server

AIFlow depends on notification service as an event dispatcher. Before running AIFlow, you need to start notification server.

```
# Initialize configuration
notification config init

# Initialize database and tables
notification db init

# Start notification server as a daemon
notification server start -d &
```

Starting AIFlow Server

```
# Initialize configuration
aiflow config init

# Initialize database and tables
aiflow db init

# Start AIFlow server as a daemon
aiflow server start -d &
```

Note: You may run into issues caused by different operating systems or versions, please refer to Troubleshooting section to get solutions.

Running a Workflow

Defining a Workflow

Below is a typically event-driven workflow. The workflow contains 4 tasks, task3 is started once both task1 and task2 finished, then task3 will send a custom event which would trigger task4 to start running.

```
import time

from ai_flow.model.action import TaskAction
from ai_flow.notification.notification_client import AIFlowNotificationClient
from ai_flow.operators.bash import BashOperator
from ai_flow.operators.python import PythonOperator
from ai_flow.model.workflow import Workflow

EVENT_KEY = "key"


def func():
    time.sleep(5)
    notification_client = AIFlowNotificationClient("localhost:50052")
    print(f"Sending event with key: {EVENT_KEY}")
    notification_client.send_event(key=EVENT_KEY,
                                   value='This is a custom message.')


with Workflow(name='quickstart_workflow') as w1:
    task1 = BashOperator(name='task1', bash_command='echo I am the 1st task.')
    task2 = BashOperator(name='task2', bash_command='echo I am the 2nd task.')
    task3 = PythonOperator(name='task3', python_callable=func)
    task4 = BashOperator(name='task4', bash_command='echo I am the 4th task.')

    task3.start_after([task1, task2])

    task4.action_on_event_received(action=TaskAction.START, event_key=EVENT_KEY)
```

You can save the above workflow as a python file on your workstation and remember the file path as ${path_of_the_workflow_file}.

Uploading the Workflow

Now you can upload the workflow with the path of the file you just saved.

```
aiflow workflow upload ${path_of_the_workflow_file}
```

You can view the workflow you uploaded by the following command:

```
aiflow workflow list --namespace default
```

Starting an Execution

The workflow you uploaded can be executed as an instance which is called execution. You can start a new execution by the following command:

```
aiflow workflow-execution start quickstart_workflow --namespace default
```

Viewing the Results

You can view the workflow execution you just started by the following command:

```
aiflow workflow-execution list quickstart_workflow --namespace default
```

The result shows `id`, `status` and other information of the workflow execution. If it is the first time you execute a workflow, the id of the workflow execution should be `1`, so you can then list tasks of workflow execution with id `1` by the following command:

```
aiflow task-execution list 1
```

Also you can check the log under `${AIFLOW_HOME}/logs` to view the outputs of tasks.

What's Next?

For more details about how to write your own workflow, please refer to the *tutorial* and and *concepts* document.

CHAPTER

TWO

INSTALLATION

2.1 Installing from PyPI

This page describes installations using the `ai-flow` package published in PyPI.

2.1.1 Prerequisites

AIFlow is tested with:

- Python: 3.7, 3.8
- Pip: 19.0.0+
- SQLite: 3.15.0+

Note: SQLite is only used in tests and getting started. To use AIFlow in production, please *set up MySQL as the backend*.

2.1.2 Installing AIFlow

Preparing Environment [Optional]

To avoid dependencies conflict, we strongly recommend using venv or other similar tools for an isolated Python environment like below:

```
python3 -m venv venv_for_aiflow
source venv_for_aiflow/bin/activate
```

Now you can install the latest AIFlow package by running:

```
python3 -m pip install ai-flow-nightly
```

Congrats, you are ready to run AIFlow and try core features following the *quickstart*.

2.1.3 Extra Dependencies

The `ai-flow-nightly` PyPI basic package only installs what's needed to get started. Additional packages can be installed depending on what will be useful in your environment. For instance, when you are setting MySQL as the metadata backend, you need to install mysqlclient by following command:

```
python -m pip install 'ai-flow-nightly[mysql]'
```

For the list of the extras and what they enable, see: *Reference for package extras*.

2.2 Installing from Sources

This page describes installations from ai-flow source code.

2.2.1 Prerequisites

Please make sure you have below tools installed on your workflow station.

- Git
- Python: 3.7, 3.8
- Pip: 19.0.0+
- SQLite: 3.15.0+

2.2.2 Preparing Environment [Optional]

To avoid dependencies conflict, we strongly recommend using venv or other similar tools for an isolated Python environment like below:

```
python3 -m venv venv_for_aiflow
source venv_for_aiflow/bin/activate
```

2.2.3 Installing wheel

AIFlow would add some entrypoints to `PATH` during installation, which requires package `wheel` installed.

```
python3 -m pip install wheel
```

2.2.4 Downloading Source Code

```
git clone https://github.com/flink-extended/ai-flow.git
```

2.2.5 Installing AIFlow

Now you can install AIFlow by running:

```
# cd into the source code directory you just cloned
cd ai-flow

# install notification service
python3 -m pip install lib/notification_service

# install ai-flow
python3 -m pip install .
```

CHAPTER

THREE

TUTORIAL AND EXAMPLES

3.1 Tutorial

This tutorial will show you how to create and run a workflow using AIFlow SDK and walk you through the fundamental AIFlow concepts and their usage. In the tutorial, we will write a simple machine learning workflow to train a Logistic Regression model and verify the effectiveness of the model using MNIST dataset.

3.1.1 Example Workflow definition

```
import logging
import os
import shutil
import time
import numpy as np

from typing import List
from joblib import dump, load

from sklearn.utils import check_random_state
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

from ai_flow import ops
from ai_flow.model.action import TaskAction
from ai_flow.operators.python import PythonOperator
from ai_flow.model.workflow import Workflow
from ai_flow.notification.notification_client import AIFlowNotificationClient,␣
↪ListenerProcessor, Event


NOTIFICATION_SERVER_URI = "localhost:50052"

current_dir = os.path.dirname(__file__)
dataset_path = os.path.join(current_dir, 'dataset', 'mnist_{}.npz')
working_dir = os.path.join(current_dir, 'tmp')

trained_model_dir = os.path.join(working_dir, 'trained_models')
validated_model_dir = os.path.join(working_dir, 'validated_models')
deployed_model_dir = os.path.join(working_dir, 'deployed_models')
```

(continues on next page)

(continued from previous page)

```
def _prepare_working_dir():
    for path in [trained_model_dir, validated_model_dir, deployed_model_dir]:
        if not os.path.isdir(path):
            os.makedirs(path)


def _get_latest_model(model_dir) -> str:
    file_list = os.listdir(model_dir)
    if file_list is None or len(file_list) == 0:
        return None
    else:
        file_list.sort(reverse=True)
        return os.path.join(model_dir, file_list[0])


def _preprocess_data(dataset_uri):
    with np.load(dataset_uri) as f:
        x_data, y_data = f['x_train'], f['y_train']

    random_state = check_random_state(0)
    permutation = random_state.permutation(x_data.shape[0])
    x_train = x_data[permutation]
    y_train = y_data[permutation]

    reshaped_x_train = x_train.reshape((x_train.shape[0], -1))
    scaler_x_train = StandardScaler().fit_transform(reshaped_x_train)
    return scaler_x_train, y_train


def preprocess():
    _prepare_working_dir()
    train_dataset = dataset_path.format('train')
    try:
        event_sender = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        while True:
            x_train, y_train = _preprocess_data(train_dataset)
            np.save(os.path.join(working_dir, f'x_train'), x_train)
            np.save(os.path.join(working_dir, f'y_train'), y_train)
            event_sender.send_event(key="data_prepared", value=None)
            time.sleep(30)
    finally:
        event_sender.close()


def train():
    """
    See also:
        https://scikit-learn.org/stable/auto_examples/linear_model/plot_sparse_
↪logistic_regression_mnist.html
    """
    _prepare_working_dir()
    clf = LogisticRegression(C=50. / 5000, penalty='l1', solver='saga', tol=0.1)
    x_train = np.load(os.path.join(working_dir, f'x_train.npy'))
    y_train = np.load(os.path.join(working_dir, f'y_train.npy'))
    clf.fit(x_train, y_train)
    model_path = os.path.join(trained_model_dir, time.strftime("%Y%m%d%H%M%S", time.
↪localtime()))
```

(continues on next page)

(continued from previous page)

```
    dump(clf, model_path)


def validate():
    _prepare_working_dir()

    validate_dataset = dataset_path.format('evaluate')
    x_validate, y_validate = _preprocess_data(validate_dataset)

    to_be_validated = _get_latest_model(trained_model_dir)
    clf = load(to_be_validated)
    scores = cross_val_score(clf, x_validate, y_validate, scoring='precision_macro')
    try:
        event_sender = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        deployed_model = _get_latest_model(deployed_model_dir)
        if deployed_model is None:
            logging.info(f"Generate the 1st model with score: {scores}")
            shutil.copy(to_be_validated, validated_model_dir)
            event_sender.send_event(key="model_validated", value=None)
        else:
            deployed_clf = load(deployed_model)
            old_scores = cross_val_score(deployed_clf, x_validate, y_validate,␣
↪scoring='precision_macro')
            if np.mean(scores) > np.mean(old_scores):
                logging.info(f"A new model with score: {scores} passes validation")
                shutil.copy(to_be_validated, validated_model_dir)
                event_sender.send_event(key="model_validated", value=None)
            else:
                logging.info(f"New generated model with score: {scores} is worse "
                             f"than the previous: {old_scores}, ignored.")
    finally:
        event_sender.close()


def deploy():
    _prepare_working_dir()
    to_be_deployed = _get_latest_model(validated_model_dir)
    deploy_model_path = shutil.copy(to_be_deployed, deployed_model_dir)
    try:
        event_sender = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        event_sender.send_event(key="model_deployed", value=deploy_model_path)
    finally:
        event_sender.close()


class ModelLoader(ListenerProcessor):
    def __init__(self):
        self.current_model = None
        logging.info("Waiting for the first model deployed...")

    def process(self, events: List[Event]):
        for e in events:
            self.current_model = e.value


def predict():
    _prepare_working_dir()
```

(continues on next page)

(continued from previous page)

```
    predict_dataset = dataset_path.format('predict')
    result_path = os.path.join(working_dir, 'predict_result')
    x_predict, _ = _preprocess_data(predict_dataset)

    model_loader = ModelLoader()
    current_model = model_loader.current_model
    try:
        event_listener = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        event_listener.register_listener(listener_processor=model_loader,
                                         event_keys=["model_deployed", ])
        while True:
            if current_model != model_loader.current_model:
                current_model = model_loader.current_model
                logging.info(f"Predicting with new model: {current_model}")
                clf = load(current_model)
                result = clf.predict(x_predict)
                with open(result_path, 'a') as f:
                    f.write(f'model [{current_model}] predict result: {result}\n')
            time.sleep(5)
    finally:
        event_listener.close()


with Workflow(name="online_machine_learning") as workflow:

    preprocess_task = PythonOperator(name="pre_processing",
                                     python_callable=preprocess)

    train_task = PythonOperator(name="training",
                                python_callable=train)

    validate_task = PythonOperator(name="validating",
                                   python_callable=validate)

    deploy_task = PythonOperator(name="deploying",
                                 python_callable=deploy)

    predict_task = PythonOperator(name="predicting",
                                  python_callable=predict)

    train_task.action_on_event_received(action=TaskAction.START, event_key="data_
↪prepared")

    validate_task.start_after(train_task)

    deploy_task.action_on_event_received(action=TaskAction.START, event_key="model_
↪validated")
```

The above Python script declares a Workflow that consists of 5 batch or streaming tasks related to machine learning. The general logic of the workflow is as follows:

1. A `pre_processing` task continuously generates training data and do some transformations. Once a batch of data is prepared, it sends an event with key `data_prepared`.
2. A `training` task starts as long as the scheduler receives an event with key `data_prepared`, the task trains a new model with the latest dataset.
3. A `validating` task starts after the `training` task finishes with status `SUCCEED` and does the model vali-

dation. If the new model is better than the deployed one, it will send an event with key `model_validated`.

4. A `deploying` task starts as long as the scheduler receives an event with key `model_validated`, the task deploys the latest model to online serving and send an event with key `model_deployed`.
5. A `predicting` task keeps running and listening to the events with key `model_deployed`, it would predict with the new deployed model as long as receiving the event.

3.1.2 Writing the Workflow

Now let us write the above workflow step by step.

As we mentioned in the *Workflow concept*, we need to write a Python script to act as a configuration file specifying the Workflow's structure. Currently, the workflow needs to contain all user-defined classes and functions in the same Python file to avoid dependency conflicts because AIFlow need to compile the Workflow object in AIFlow server and workers.

Importing Modules

As the workflow is defined in a Python script, we need to import the libraries we need.

Note: The libraries that we imports need to be installed on AIFlow server and workers in advance to avoid importing error.

```
import logging
import os
import shutil
import time
import numpy as np

from typing import List
from joblib import dump, load

from sklearn.utils import check_random_state
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression

from ai_flow import ops
from ai_flow.model.action import TaskAction
from ai_flow.operators.python import PythonOperator
from ai_flow.model.workflow import Workflow
from ai_flow.notification.notification_client import AIFlowNotificationClient,␣
↪ListenerProcessor, Event
```

Defining the Workflow

A Workflow is declared in a `with` statement, which includes all Tasks inside it. When you initialize the Workflow, you need to give it a name(required) and a *namespace*(optional). If no namespace is assigned, the workflow belongs to `default` namespace.

In the example, we create a workflow named `online_machine_learning`, belonging to `defalut` namespace.

```
with Workflow(name="online_machine_learning") as workflow:
    ...
```

Now let us define the AIFlow Tasks, note that the tasks defined in the workflow will run on different workers at different points in time, so no variables in memory should be passed between them to cross communicate.

Defining the preprocessing Task

Here we create a `PythonOperator` that accepts a function as a parameter to preprocess dataset before training. As we mentioned in the *Operator concept*, an Operator that is instantiated can be called Task, so we could say that we create a Task named `preprocessing` in Workflow `online_machine_learning`.

Note: The definition of the Task should always be under the `with` statement of the Workflow that contains it.

We use a while loop to simulate continuous data generation and transformation. In each loop, we transform the dataset with sklearn API and save the new dataset to local file, then we send an Event with `AIFlowNotificationClient` to notify that a new batch of data has been prepared.

```
with Workflow(name="online_machine_learning") as workflow:
    preprocess_task = PythonOperator(name="pre_processing",
                                     python_callable=preprocess)


def _prepare_working_dir():
    for path in [trained_model_dir, validated_model_dir, deployed_model_dir]:
        if not os.path.isdir(path):
            os.makedirs(path)


def _preprocess_data(dataset_uri):
    with np.load(dataset_uri) as f:
        x_data, y_data = f['x_train'], f['y_train']

    random_state = check_random_state(0)
    permutation = random_state.permutation(x_data.shape[0])
    x_train = x_data[permutation]
    y_train = y_data[permutation]

    reshaped_x_train = x_train.reshape((x_train.shape[0], -1))
    scaler_x_train = StandardScaler().fit_transform(reshaped_x_train)
    return scaler_x_train, y_train


def preprocess():
    _prepare_working_dir()
    train_dataset = dataset_path.format('train')
    try:
```

(continues on next page)

(continued from previous page)

```
        event_sender = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        while True:
            x_train, y_train = _preprocess_data(train_dataset)
            np.save(os.path.join(working_dir, f'x_train'), x_train)
            np.save(os.path.join(working_dir, f'y_train'), y_train)
            event_sender.send_event(key="data_prepared", value=None)
            time.sleep(30)
    finally:
        event_sender.close()
```

Defining the training Task

The `training` task loads the dataset that is preprocessed and trains a model with Logistic Regression algorithm, and then save the model to the local directory `trained_models`. The `training` task has a *Task Rule* declared by `action_on_event_received` API, which means that the `training` task takes the action `START` as long as an event with key `data_prepared` happened.

```
with Workflow(name="online_machine_learning") as workflow:
    train_task = PythonOperator(name="training",
                                python_callable=train)
    train_task.action_on_event_received(action=TaskAction.START, event_key="data_
↪prepared")


def train():
    _prepare_working_dir()
    clf = LogisticRegression(C=50. / 5000, penalty='l1', solver='saga', tol=0.1)
    x_train = np.load(os.path.join(working_dir, f'x_train.npy'))
    y_train = np.load(os.path.join(working_dir, f'y_train.npy'))
    clf.fit(x_train, y_train)
    model_path = os.path.join(trained_model_dir, time.strftime("%Y%m%d%H%M%S", time.
↪localtime()))
    dump(clf, model_path)
```

Defining the validating Task

The `validating` task loads and proprocess the validation dataset and score the latest model with cross validation. If the score of the new trained model is better than the current deployed one, send an event with key `model_validated` to notify that a better model is generated.

The `validating` task also has a *Task Rule* which is declared by `start_after` API, which means that the `validating` starts right after the `training` succeeds.

```
with Workflow(name="online_machine_learning") as workflow:
    validate_task = PythonOperator(name="validating",
                                   python_callable=validate)
    validate_task.start_after(train_task)


def validate():
    _prepare_working_dir()

    validate_dataset = dataset_path.format('evaluate')
```

(continues on next page)

(continued from previous page)

```
    x_validate, y_validate = _preprocess_data(validate_dataset)

    to_be_validated = _get_latest_model(trained_model_dir)
    clf = load(to_be_validated)
    scores = cross_val_score(clf, x_validate, y_validate, scoring='precision_macro')
    try:
        event_sender = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        deployed_model = _get_latest_model(deployed_model_dir)
        if deployed_model is None:
            logging.info(f"Generate the 1st model with score: {scores}")
            shutil.copy(to_be_validated, validated_model_dir)
            event_sender.send_event(key="model_validated", value=None)
        else:
            deployed_clf = load(deployed_model)
            old_scores = cross_val_score(deployed_clf, x_validate, y_validate,␣
↪scoring='precision_macro')
            if np.mean(scores) > np.mean(old_scores):
                logging.info(f"A new model with score: {scores} passes validation")
                shutil.copy(to_be_validated, validated_model_dir)
                event_sender.send_event(key="model_validated", value=None)
            else:
                logging.info(f"New generated model with score: {scores} is worse "
                             f"than the previous: {old_scores}, ignored.")
    finally:
        event_sender.close()
```

Defining the deploying Task

The `deploying` task simulates the deployment by copying the model from the directory `validated_models` to `deployed_models`. After deploying the model, the task will send an event with key `model_deployed` to notify that the new model has been deployed.

The `deploying` task also has a *Task Rule* which is declared by `action_on_event_received` API, which means that the `deploying` starts as long as an event with key `model_validated` happened.

```
with Workflow(name="online_machine_learning") as workflow:
    deploy_task = PythonOperator(name="deploying",
                                 python_callable=deploy)
    deploy_task.action_on_event_received(action=TaskAction.START, event_key="model_
↪validated")


def deploy():
    _prepare_working_dir()
    to_be_deployed = _get_latest_model(validated_model_dir)
    deploy_model_path = shutil.copy(to_be_deployed, deployed_model_dir)
    try:
        event_sender = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        event_sender.send_event(key="model_deployed", value=deploy_model_path)
    finally:
        event_sender.close()
```

Defining the predicting Task

In the `predicting` task, we create a custom event listener to keep listening to events with key `model_deployed`, when it receives the event, it will predict with the latest deployed model. The `predicting` task has no *Task Rules* so it will start as long as the workflow begins.

```
class ModelLoader(ListenerProcessor):
    def __init__(self):
        self.current_model = None
        logging.info("Waiting for the first model deployed...")

    def process(self, events: List[Event]):
        for e in events:
            self.current_model = e.value


def predict():
    _prepare_working_dir()
    predict_dataset = dataset_path.format('predict')
    result_path = os.path.join(working_dir, 'predict_result')
    x_predict, _ = _preprocess_data(predict_dataset)

    model_loader = ModelLoader()
    current_model = model_loader.current_model
    try:
        event_listener = AIFlowNotificationClient(NOTIFICATION_SERVER_URI)
        event_listener.register_listener(listener_processor=model_loader,
                                         event_keys=["model_deployed", ])
        while True:
            if current_model != model_loader.current_model:
                current_model = model_loader.current_model
                logging.info(f"Predicting with new model: {current_model}")
                clf = load(current_model)
                result = clf.predict(x_predict)
                with open(result_path, 'a') as f:
                    f.write(f'model [{current_model}] predict result: {result}\n')
            time.sleep(5)
    finally:
        event_listener.close()
```

3.1.3 Running the Example

To get the full example along with the dataset, please download them from github.

Uploading the Workflow

Now we have a complete online machine learning workflow and its required dataset. Let's upload them to AIFlow server.

```
aiflow workflow upload ${path_to_workflow_file} --files ${path_to_dataset_directory}
```

The workflow is uploaded successfully if you see `Workflow: default.online_machine_learning, submitted.` on the console.

Starting the Workflow

In AIFlow, starting a workflow is creating a new workflow execution, you can do this by the following command.

```
aiflow workflow-execution start online_machine_learning
```

The workflow execution is started if you see `Workflow execution: {} submitted.` on the console. You can view the workflow execution you just created by `list` command:

```
aiflow workflow-execution list online_machine_learning
```

Viewing the results

You can view the status of the tasks by the following command:

```
aiflow task-execution list ${workflow_execution_id}
```

Also you can view the prediction output in the file `${AIFLOW_HOME}/working_dir/online_machine_learning/*/online_ml_workflow/tmp/predict_result`

If you want to view logs, you can go to check logs under the directory `${AIFLOW_HOME}/logs/online_machine_learning/`. The log files will give you the information in detail.

Stopping the Workflow Execution

The `online_machine_learning` workflow contains streaming tasks that will never stop. If you want to stop the workflow execution, you can run the following command:

```
aiflow workflow-execution stop-all online_machine_learning
```

3.1.4 What's Next

Congratulations! You have been equipped with the necessary knowledge to write your own workflow. At this point, you can check *Examples* for more examples and *concepts* to write your own workflows.

3.2 Examples

Below, you can find a number of examples for various AIFlow use cases.

- quickstart
- FlinkOpertor
- SparkOperator
- periodic
- event triggered workflow
- custom condition

CHAPTER

FOUR

CONCEPTS

4.1 Workflows

A Workflow consists of *Tasks*, organized with *Task Rules* to describe how they should run. The Workflow and Tasks are defined in a Python script which just acts as a configuration file specifying the Workflow's structure as code.

4.1.1 Declaring Workflows

A Workflow is declared in a `with` statement, which includes all Tasks and Task Rules inside it.

```
from ai_flow.model.workflow import Workflow
from ai_flow.operators.bash import BashOperator

with Workflow(name='workflow_name') as workflow:
    task1 = BashOperator(name='task_1',
                         bash_command='echo I am the 1st task')
    task2 = BashOperator(name='task_2',
                         bash_command='echo I am the 2nd task')
    task2.start_after([task1, ])
```

AIFlow will execute the Python file and then load any Workflow objects at the *top level* in the file. This means you can define multiple Workflows per Python file.

4.1.2 Uploading Workflows

Users can upload Workflows by the command-line interface. In addition to the Python file containing the Workflow objects, other files that are used in Workflow definition and execution should also be uploaded by `--files` option.

```
aiflow workflow upload workflow.py --files f1,f2
```

4.1.3 Running Workflows

A Workflow can be executed to generate Workflow Execution. There are 3 ways to run Workflow and [generate workflow executions](./workflow_executions.md#Creating Workflow Execution).

4.1.4 Workflow disabling and deletion

A Workflow can be disabled which means no more Workflow Executions or [Task Executions](./tasks.md#Task Executions) will be scheduled.

```
aiflow workflow disable workflow_name
```

However, the disabling operation does not delete the metadata of the Workflow, users can enable the Workflow to resume the scheduling of it if needed.

```
aiflow workflow enable workflow_name
```

If you want to not only disable the workflow but also delete the metadata, please run the following command:

```
aiflow workflow delete workflow name
```

Note: The deletion command truncates all metadata of the Workflow in cascade, including Workflows, Workflow Executions and Task Executions, so before deleting the Workflow, please make sure that no executions of the Workflow is still running.

4.2 Namespaces

Namespaces provide a mechanism for isolating groups of *Workflows* within a single cluster. Names of Workflows need to be unique within a namespace, but not across Namespaces. Multiple business-related Workflows can be put into the same Namespace to have the same access control and Event isolation.

4.2.1 Creating Namespaces

AIFlow has a default namespace called `default`. Users can also create their own namespaces if needed through the command-line interface.

```
aiflow namespace add user_namespace
```

4.2.2 Viewing Namespaces

```
aiflow namespace list
```

4.2.3 Deleting Namespaces

```
aiflow namespace delete user_namespace
```

4.3 Tasks

A Task is the basic unit of execution in *Workflow*. Tasks are arranged into a Workflow, and they have *Task Rules* between them in order to describe the conditions under which they should run.

4.3.1 Task Executions

Much in the same way that a *Workflow* is instantiated into a Workflow Execution each time it runs, the tasks are instantiated into *Task Executions*.

A Task Execution has a Status representing what stage of the lifecycle it is in. The possible Status for a Task Execution is:

- init: The task execution has not yet been queued (its dependencies are not yet met)
- queued: The task execution has been assigned to an Executor and is awaiting a worker
- running: The task execution is running on a worker
- success: The task execution finished running without errors
- failed: The task execution had an error during execution and failed to run
- stopping: The task execution was externally requested to shut down when it was running, but not yet finish stopping
- stopped: The task execution is requested to shut down and successfully stopped
- retrying: The task execution failed, but has retry attempts left and will be rescheduled.

Note: In a Workflow Execution, there can be only one running execution of each task, nothing would happen even if you force start a running task.

4.3.2 Task Actions

A Task can perform different actions according to the *Task Rule*. There are three kinds of actions of a task.

- start: Start a new Task Execution if there is no running execution, otherwise do nothing.
- stop: Stop a running Task Execution.
- restart: Stop the currently running Task Execution and start a new execution.

4.4 Operators

An Operator is conceptually a template for a predefined *Task*, in other words, Task is an instantiated Operator. AIFlow has an extensive set of operators available and some popular operators are built-in to the core:

- BashOperator - executes a bash command
- PythonOperator - calls an arbitrary Python function
- FlinkOperator - executes a `flink run` command to submit various Flink job
- SparkOperator - executes a `spark-submit` or `spark-sql` command to run various Spark job

4.4.1 Operator Config

AIFlow Operators have some common configurations that can be passed as parameters when initializing the Operator.

Periodic Task

Similar to Workflow, A Task can also run periodically by passing parameters `periodic_expression`. Instead of binding to a *Workflow Schedule*, A Task can only have one periodic expression which has the same format as the *Workflow Schedule*, e.g.

```
from ai_flow.model.workflow import Workflow
from ai_flow.operators.bash import BashOperator

with Workflow(name='periodic_task_example') as workflow:
    task1 = BashOperator(name='task_1',
                         bash_command='echo I am the 1st task',
                         periodic_expression='cron@*/1 * * * *')
    task2 = BashOperator(name='task_2',
                         bash_command='echo I am the 2nd task',
                         periodic_expression='interval@0 0 1 0')
    task3 = BashOperator(name='task_3',
                         bash_command='echo I am the 3rd task')
    task3.start_after([task1, ])
```

Note: As AIFlow is event-based, tasks who start after a periodic task will also run periodically right after the upstream task finishes. In the above example, task3 will start running every time task1 finished.

4.5 Task Rules

A Task/Operator usually has some rules which describe when and how it should take action. A Task Rule consists of three parts:

- *Event* - it specifies the signal that triggers the invocation of the rule
- *Condition* - it is a logical test that, if satisfied or evaluates to be true, causes the action to be carried out
- Action - START, STOP or RESTART the task

In a Workflow, those Tasks that do not have rules that Action is `START` will be executed as long as the Workflow starts. During the execution of those Tasks that run first, some Events would be generated to trigger the other Tasks to run.

Next, we will go deep into some types of Rules to help thoroughly understand them.

4.5.1 Status Rules

The most common Task Rule is that one task runs after the other tasks succeed, users can add such Rules by calling `start_after` API.

```
from ai_flow.model.workflow import Workflow
from ai_flow.operators.bash import BashOperator

with Workflow(name='my_workflow') as workflow:
    task1 = BashOperator(name='task_1',
                         bash_command='echo I am the 1st task')
    task2 = BashOperator(name='task_2',
                         bash_command='echo I am the 2nd task')
    task2.start_after([task1, ])
```

`task1` has no Rules that Action is `START` so it would execute first, and `task2` will start running after `task1` succeed.

More generally, a task may perform other actions after more than one task is finished with any status, users can add such Rules by calling `action_on_task_status` API.

```
from ai_flow.model.action import TaskAction
from ai_flow.model.status import TaskStatus
from ai_flow.model.workflow import Workflow
from ai_flow.operators.bash import BashOperator

with Workflow(name='my_workflow') as workflow:
    task1 = BashOperator(name='task_1',
                         bash_command='sleep 10')
    task2 = BashOperator(name='task_2',
                         bash_command='sleep 20')
    task3 = BashOperator(name='task_3',
                         bash_command='sleep 100')
    task3.action_on_task_status(action=TaskAction.STOP,
                                upstream_task_status_dict={
                                    task1: TaskStatus.SUCCESS,
                                    task2: TaskStatus.SUCCESS
                                })
```

Since all 3 tasks have no Rules that Action is `START` so they will execute once the workflow starts, but `task3` won't finish after 100 seconds, instead it will be stopped when both `task1` and `task2` succeed.

4.5.2 Single Event Rules

Another commonly used Task Rule is that one task takes actions after receiving an event, users can add such Rules by calling `action_on_event_received` API.

```
import time

from ai_flow.model.action import TaskAction
from ai_flow.model.workflow import Workflow
from ai_flow.notification.notification_client import AIFlowNotificationClient
from ai_flow.operators.bash import BashOperator
from ai_flow.operators.python import PythonOperator

def func():
    time.sleep(5)
    print('I am the 1st task')
    notification_client = AIFlowNotificationClient("localhost:50052")
    notification_client.send_event(key="key",
                                   value="")

with Workflow(name='quickstart_workflow') as workflow:
    task1 = PythonOperator(name='task1', python_callable=func)
    task2 = BashOperator(name='task2', bash_command='echo I am the 2nd task.')
    task2.action_on_event_received(action=TaskAction.START, event_key="key")
```

`task1` would send a custom event that triggers `task2` to start running.

4.5.3 Custom Rules

Sometimes users may want to take action on tasks only when they receive multiple events or satisfy more complex conditions. In those scenarios, users can add custom Task Rules by calling `action_on_condition` API, e.g. in the below example, `task1` sends an event with a number and `task2` would be triggered when the number adds up to 100.

```
import random
import time

from notification_service.model.event import Event

from ai_flow.model.action import TaskAction
from ai_flow.model.condition import Condition
from ai_flow.model.context import Context
from ai_flow.model.state import ValueState, ValueStateDescriptor
from ai_flow.model.workflow import Workflow
from ai_flow.notification.notification_client import AIFlowNotificationClient
from ai_flow.operators.bash import BashOperator
from ai_flow.operators.python import PythonOperator

class NumCondition(Condition):
    def is_met(self, event: Event, context: Context) -> bool:
        state: ValueState = context.get_state(ValueStateDescriptor(name='num_state'))
        num = 0 if state.value() is None else int(state.value())
        num += int(event.value)
        if num >= 100:
            return True
        else:
```

(continues on next page)

(continued from previous page)

```
            state.update(num)
            return False

def random_produce():
    notification_client = \
        AIFlowNotificationClient(server_uri='localhost:50052')
    while True:
        num = random.randint(0, 9)
        notification_client.send_event(key='num_event', value=str(num))
        time.sleep(1)

with Workflow(name='condition_workflow') as workflow:
    task1 = PythonOperator(name='producer',
                           python_callable=random_produce)
    task2 = BashOperator(name='consumer',
                         bash_command='echo Got 100 records.')

    task2.action_on_condition(action=TaskAction.START,
                              condition=NumCondition(expect_event_keys=['num_event']))
```

4.6 Conditions

A Condition is a logical test that, if satisfied or evaluates to be true, causes the action to be carried out.

4.6.1 When to evaluate

A Condition consists of a list of expected keys of *Events* and a logical test, only when one of the expected *Events* comes, the logical test will be evaluated.

4.6.2 Custom Condition

It is allowed to define custom Conditions according to various scenarios by inheriting class `Condition` and implementing `is_met` function, e.g.

```
from notification_service.model.event import Event

from ai_flow.model.condition import Condition
from ai_flow.model.context import Context
from ai_flow.model.state import ValueState, ValueStateDescriptor

class NumCondition(Condition):
    def is_met(self, event: Event, context: Context) -> bool:
        state: ValueState = context.get_state(ValueStateDescriptor(name='num_state'))
        num = 0 if state.value() is None else int(state.value())
        num += int(event.value)
        if num >= 100:
            return True
        else:
            state.update(num)
            return False
```

The above examples shows a Condition that is satisfied only when it receives enough events that the number adds up to 100. With the `NumCondition`, we can easily define a Workflow that the consumer task starts only when the upstream producers prepared more than 100 records.

```
import random
import time

from ai_flow.model.action import TaskAction
from ai_flow.model.workflow import Workflow
from ai_flow.notification.notification_client import AIFlowNotificationClient
from ai_flow.operators.bash import BashOperator
from ai_flow.operators.python import PythonOperator

def random_produce():
    notification_client = \
        AIFlowNotificationClient(server_uri='localhost:50052')
    while True:
        num = random.randint(0, 9)
        notification_client.send_event(key='num_event', value=str(num))
        time.sleep(1)

with Workflow(name='condition_workflow') as workflow:
    task1 = PythonOperator(name='producer',
                           python_callable=random_produce)
    task2 = BashOperator(name='consumer',
                         bash_command='echo Got 100 records.')

    task2.action_on_condition(action=TaskAction.START,
                              condition=NumCondition(expect_event_keys=['num_event']))
```

4.7 Events

The event specifies the signal that triggers evaluating *Condition* and taking the action. AIFlow scheduler relies on internal events to decide which Workflow and Tasks to perform actions. Users can also send custom Events in Tasks, there are three main uses of custom Events:

- Trigger a *Workflow Trigger*.
- Trigger a *Task Rule*.
- Transfer messages between Tasks in the same namespace.

4.7.1 Sending Events

A user Event is sent with `AIFlowNotificationClient`, and passing `key` and `value` with string type as parameters. There are some design constraints to be aware of:

- The `AIFlowNotificationClient` can only be instantiated in a Task runtime.
- The Event can only be transferred in the same AIFlow *Namespace*.
- If the Event is used to trigger *Task Rules*, it can only effect on Tasks in the same Workflow Execution.

Here's an example of Tasks triggered by a custom Event.

```
from ai_flow.model.action import TaskAction
from ai_flow.notification.notification_client import AIFlowNotificationClient
from ai_flow.operators.bash import BashOperator
from ai_flow.operators.python import PythonOperator

from ai_flow.model.workflow import Workflow

def func():
    notification_client = AIFlowNotificationClient("localhost:50052")
    notification_client.send_event(key="key",
                                   value='This is a custom message.')

with Workflow(name='workflow') as workflow:
    task1 = PythonOperator(name='task1', python_callable=func)
    task2 = BashOperator(name='task2', bash_command='echo I am the 2nd task.')

    task2.action_on_event_received(action=TaskAction.START, event_key="key")
```

4.7.2 Listening Events

Users can also listen to Events with `AIFlowNotificationClient` in Tasks to receive messages from other Tasks. To listen to Events, you need to implement your own `ListenerProcessor` to define the logic of handling Events, e.g.

```
from typing import List

from ai_flow.notification.notification_client import ListenerProcessor, Event


class Counter(ListenerProcessor):
    def __init__(self):
        self.counter = 0

    def process(self, events: List[Event]):
        self.counter += len(events)
```

Then you can start listening to Events by calling `register_listener`, e.g.

```
from ai_flow.notification.notification_client import AIFlowNotificationClient

counter = Counter()
client = AIFlowNotificationClient("localhost:50052")
listener_id = client.register_listener(listener_processor=counter,
                                       event_keys=['expect_key',])
```

`register_listener` will create a new thread to listen to Events with key=`expect_key`, so please remember to call `unregister_listener` to release resources.

```
client.unregister_listener(listener_id)
```

4.8 Workflow Schedules

A Workflow Schedule is the periodic execution plan of the *Workflow*.

4.8.1 Creating Schedules

Users can add a Workflow Schedule to a Workflow by the following command:

```
aiflow workflow-schedule add workflow_name expression
```

The `expression` has two supported types: **cron** and **time interval**.

Cron

Describes when to run the Workflow with a `Cron` expression which is in the format `cron@expression`. The `expression` is a standard crontab expression, see https://en.wikipedia.org/wiki/Cron for more information on the format accepted here.

The below command adds a Workflow Schedule to `my_workflow`, which makes the Workflow run at every hour.

```
aiflow workflow-schedule add my_workflow "cron@0 * * * *""
```

Time Interval

Describes how often to run the Workflow from now on in the format `interval@days hours minutes seconds`, e.g. `interval0 0 10 0` means run the Workflow every 10 minutes from now on.

```
aiflow workflow-schedule add my_workflow "interval0 0 10 0"
```

4.8.2 Viewing Schedules

Users can view all Schedules of the Workflow by the following command:

```
aiflow workflow-schedule list my_workflow
```

4.8.3 Pausing and Resuming Schedules

If you want to temporarily stop a periodic schedule, you can run the following command:

```
aiflow workflow-schedule pause workflow_execution_id
```

Note that the above command doesn't delete the metadata of the Workflow Schedule, you can resume the periodic scheduling if needed.

```
aiflow workflow-schedule resume workflow_execution_id
```

4.8.4 Deleting Schedules

To completely delete the metadata of the Workflow Schedule, you can use the `delete` sub-command.

```
aiflow workflow-schedule delete workflow_execution_id
```

4.9 Workflow Triggers

Similar to *Task Rule*, a Workflow can also have some rules on it called **Workflow Trigger**, however, a Workflow Trigger only consists of Event and Condition. When the Event comes and the Condition is satisfied, the Workflow would be started, and no other types of Action(stop, restart) are supported.

4.9.1 Creating Workflow Triggers

User can create a Workflow Trigger by `ops.add_workflow_trigger` with a `WorkflowRule` passed, e.g. the following code makes workflow `event_triggered_workflow` execute as long as received an event with key `trigger_workflow`.

```
from ai_flow import ops
from ai_flow.model.internal.conditions import SingleEventCondition
from ai_flow.model.rule import WorkflowRule
from ai_flow.model.workflow import Workflow
from ai_flow.notification.notification_client import AIFlowNotificationClient
from ai_flow.operators.bash import BashOperator
from ai_flow.operators.python import PythonOperator

def send_event():
    client = AIFlowNotificationClient(server_uri='localhost:50052')
    client.send_event(key='trigger_workflow', value=None)


with Workflow(name='event_trigger_workflow_1') as w1:
    event_task = PythonOperator(name='event_task',
                                python_callable=send_event)

with Workflow(name='event_trigger_workflow_2') as w2:
    task1 = BashOperator(name='task1',
                         bash_command='echo I am 1st task.')


if __name__ == "__main__":
    ops.upload_workflows(__file__)
    trigger_rule = WorkflowRule(SingleEventCondition(expect_event_key="trigger_
↪workflow"))
    ops.add_workflow_trigger(rule=trigger_rule, workflow_name='event_trigger_workflow_
↪2')
    ops.start_workflow_execution('event_trigger_workflow_1')
```

```
Currently, only Python API is supported to create Workflow Trigger.
```

4.9.2 Viewing Triggers

Users can view all Workflow Triggers of the Workflow by the following command:

```
aiflow workflow-trigger list workflow_name
```

4.9.3 Pausing and Resuming Triggers

If you want to temporarily stop a Workflow Trigger, you can run the following command.

```
aiflow workflow-trigger pause workflow_trigger_id
```

Note that the above command doesn't delete the metadata of the Workflow Trigger, you can resume the trigger if needed.

```
aiflow workflow-trigger resume workflow_trigger_id
```

4.9.4 Deleting Triggers

To completely delete the metadata of the Workflow Trigger, you can use the `delete` sub-command.

```
aiflow workflow-trigger delete workflow_trigger_id
```

CHAPTER

FIVE

OPERATION

5.1 Deploying Notification Server

AIFlow relies on a notification service to handle event dispatching and listening. The notification service could be any message queue that complies with the AIFlow specification. AIFlow provides an embedded implementation which is lightweight, exactly-once and highly available.

In this guide, we will demonstrate how to deploy a Notification Server.

5.1.1 Installation

Before deploying, please make sure you have followed the *Installation Guide* to install Notification Service and AIFlow.

5.1.2 Initialize Configuration

To initialize the default configuration file, you can run the following command:

```
notification config init
```

This command will generate the default configuration file `notification_server.yaml` in the `$NOTIFICATION_HOME` directory(`$HOME/notification_service` by default).

Note: If the configration file already exists, the command will not generate the configration file any more. If you want to reset the configration, you need to remove it manually and then run the script again.

If you want to learn all configurations, you can refer to *here*.

5.1.3 Initialize Database

The database uri of Notification Server is configured in `notification_server.yaml`, you can run following command to initialize the database configured.

```
notification db init
```

5.1.4 Start the Notification Server

You can start the Notification Server with the following command in daemon mode.

```
notification server start -d
```

It will start the Notification Server in a background process. You can check the log of the Notification Server at `$NOTIFICATION_HOME/logs` directory. `notification_server-*.log` is the log of Notification Server. If you see " notification server started." in the log, the Notification Server successfully started.

5.1.5 Configuration

This section shows an exhaustive list of available configuration of the Notification Server.

Notification Server

Key	Type	Default	Description
server_port	Integer	50052	The port where the Notification Server is exposed.
db_uri	String	sqlite:///${NOTIFICATION_HOME}/ns.db	The uri of the database backend for Notification Server.
enable_ha	String	False	Whether to start server in HA mode.
ha_ttl_ms	Integer	10000	The time in millisecond to detect living members in HA mode.
advertised_uri	String	localhost:50052	Uri of server registered in HA manager for clients to use.
wait_for_server_started_timeout	Double	5.0	timeout for notification server to be available after started in seconds.

5.1.6 Default Notification Server Configuration

```
# port of notification server
server_port: 50052
# uri of database backend for notification server
db_uri: sqlite:///${NOTIFICATION_HOME}/ns.db
```

Note: The variable `${NOTIFICATION_HOME}` in above configuration should be replaced with your own path.

5.2 Deploying AIFlow Server

In this guide, we demonstrate how to deploy an AIFlow Server.

5.2.1 Initialize Configuration

To initialize the default configuration file, you can run the following command:

```
aiflow config init
```

This command will generate the default configuration file `aiflow_server.yaml` in the `$AIFLOW_HOME` directory(`$HOME/aiflow` by default).

Note: If the config file already exists, the command will not generate the default configuration. If you want to reset the configration, you need to remove it manually and then run the script again.

If you want to learn all configurations, you can refer to *here*.

5.2.2 Initialize Database

The database uri of AIFlow Server is configured in `aiflow_server.yaml`, you can run following command to initialize database.

```
aiflow db init
```

5.2.3 Start the AIFlow Server

Note: AIFlow Server requires Notification Server to work. Please make sure you have deployed a notification server and configure the notification uri in the AIFlow Server config file accordingly.

You can start the AIFlow Server with the following commands.

```
aiflow server start -d
```

It will start the AIFlow Server in background processes. You can check the log at `$AIFLOW_HOME/logs` directory.

5.2.4 Configuration

This section shows an exhaustive list of available configuration of the AIFlow Server.

AIFlow server

Key	Type	Default	Description
log_dir	String	${AIFLOW_HOME}	The base log folder of the scheduler and job executions.
rpc_port	Integer	50051	The rpc port where the AIFlow server is exposed to client.
internal_rpc_port	Integer	50000	The rpc port where the AIFlow server exposed for internal communication.
rest_port	Integer	8000	The port where the AIFlow rest server exposed.
metadata_backend_uri	String	sqlite:///${AIFLOW_HOME}/aiflow.db	The uri of the database backend for AIFlow Server.
state_backend_uri	String	sqlite:///${AIFLOW_HOME}/aiflow.db	The uri of the state backend.
sql_alchemy_pool_enabled	Boolean	True	Whether SqlAlchemy enables pool database connections.
sql_alchemy_pool_size	Integer	5	The maximum number of database connections in the pool. 0 indicates no limit.
sql_alchemy_max_overflow	Integer	10	The maximum overflow size of the pool.
history_retention	String	30d	Metadata and log history retention.
notification_server_uri	String	127.0.0.1:50052	The uri of the Notification Server that the AIFlow Server connect to.

Task Executor

Key	Type	Default	Description
task_executor	String	Local	The executor to run tasks, options: local, kubernetes
task_executor_heartbeat_check_interval	Integer	10	The interval in seconds that the task executor check the heartbeat of task executions.
task_heartbeat_interval	Integer	10	The interval in seconds that the task executions send heartbeats.
task_heartbeat_timeout	Integer	60	The timeout in seconds that the task executions is treated as timeout.
local_executor_parallelism	Integer	10	Num of workers of local task executor.

5.2.5 Default AIFlow server Configuration

```
# directory of AIFlow logs
log_dir : {AIFLOW_HOME}/logs

# port of rpc server
rpc_port: 50051

# port of internal rpc
internal_rpc_port: 50000
```

(continues on next page)

(continued from previous page)

```
# port of rest server
rest_port: 8000

# uri of database backend for AIFlow server
metadata_backend_uri: sqlite:///{AIFLOW_HOME}/aiflow.db

# metadata and log history retention
history_retention: 30d

# uri of state backend
state_backend_uri: sqlite:///{AIFLOW_HOME}/aiflow.db

# whether SqlAlchemy enables p s.
sql_alchemy_pool_enabled: True

# the maximum number of database connections in the pool. 0 indicates no limit.
sql_alchemy_pool_size: 5

# the maximum overflow size of the pool.
sql_alchemy_max_overflow: 10

# uri of the server of notification service
notification_server_uri: 127.0.0.1:50052

# task executor, options: local, kubernetes
task_executor: Local

# the interval in seconds that the task executor check the heartbeat of task␣
↪executions
task_executor_heartbeat_check_interval: 10

# the timeout in seconds that the task executions is treated as timeout
task_heartbeat_timeout: 60

# the interval in seconds that the task executions send heartbeats
task_heartbeat_interval: 10

# num of workers of local task executor
local_executor_parallelism: 10

# kubernetes task executor config
k8s_executor_config:
  pod_template_file:
  image_repository:
  image_tag:
  namespace:
  in_cluster: False
  kube_config_file:
```

Note: The variable `${AIFLOW_HOME}` in above configuration should be replaced with your own path.

5.3 Client Configurations

As a client-server application, AIFlow allows users to access the server from any network connected machine. That means you can upload and manage the workflow from any client. An AIFlow client needs a configuration file `aiflow_client.yaml` under ${AIFLOW_HOME}. Here are the configurations of the `aiflow_client.yaml`.

Key	Type	Default	Description
server_address	String	localhost:50051	The uri of the AIFlow server.
blob_manager_class	String	ai_flow.blob_manager.impl.local_blob_manager.LocalBlobManager	The fully qualified name of the `BlobManager` class.
blob_manager_config	dict	None	Custom configuration of this type of implementation.

For the full blob manager config, please refer to *here*

5.3.1 Default AIFlow server Configuration

```yaml
# address of AIFlow server
server_address: localhost:50051

# configurations about blob manager
blob_manager:
  blob_manager_class: ai_flow.blob_manager.impl.local_blob_manager.LocalBlobManager
  blob_manager_config:
    root_directory: {AIFLOW_HOME}/blob
```

CHAPTER

SIX

PLUGINS

6.1 Blob Manager Plugin

`Blob Managers` are the central storage that supports uploading and downloading files. There are four purposes of having them:

- The AIFlow client needs to submit artifacts(user codes, dependencies, and resources) to AIFlow server.
- The AIFlow Server needs to distribute artifacts among workers.
- The artifacts of each execution should be stored in persistent storage for restoring.
- Users may need to transfer files between jobs in the same project.

`Blob Managers` have a common API and are "pluggable", meaning you can swap `Blob Manager` based on your needs. AIFlow provides some built-in implementations, you can choose one of them or even implement your own `BlobManager` if needed.

Each project can only have one `Blob Manager` configured at a time, this is set by the `blob` section on top-level of the `project.yaml`. The `blob` section has two required sub-configs:

- blob_manager_class: the fully-qualified name of the `Blob Manager` class.
- blob_manager_config: custom configuration of this type of implementation.

6.1.1 Built-in Blob Managers

LocalBlobManager

`LocalBlobManager` is only used when the AIFlow client, server, and workers are all on the same host because it relies on the local file system. `LocalBlobManager` has following custom configurations:

Key	Type	DESCRIPTION
root_directory	String	The root directory of local filesystem to store artifacts

A complete configuration example of `LocalBlobManager` in `project.yaml`.

```
blob:
  blob_manager_class: ai_flow_plugins.blob_manager_plugins.local_blob_manager.
↪LocalBlobManager
  blob_manager_config:
    root_directory: /tmp
```

OssBlobManager

`OssBlobManager` relies on Alibaba Cloud OSS to store resources. To use `OssBlobManager` you need to install python SDK for OSS client on every node that needs to access OSS file system.

```
pip install 'ai-flow-nightly[oss]'
```

`OssBlobManager` has following custom configurations:

Key	Type	DESCRIPTION
root_directory	String	The root path of OSS filesystem to store artifacts
access_key_id	String	The id of the access key
access_key_secret	String	The secret of the access key
endpoint	String	Access domain name or CNAME
bucket	String	The name of OSS bucket

A complete configuration example of `OssBlobManager` in `project.yaml`.

```
blob:
  blob_manager_class: ai_flow_plugins.blob_manager_plugins.oss_blob_manager.
↪OssBlobManager
  blob_manager_config:
        access_key_id: xxx
        access_key_secret: xxx
        endpoint: oss-cn-hangzhou.aliyuncs.com
        bucket: ai-flow
        root_directory: tmp
```

HDFSBlobManager

`HDFSBlobManager` relies on HDFS to store resources. To use `HDFSBlobManager` you need to install python SDK for HDFS client on every node which needs to access `HDFSBlobManager`.

```
pip install 'ai-flow-nightly[hdfs]'
```

`HDFSBlobManager` has following custom configurations:

Key	Type	DESCRIPTION
hdfs_url	String	The url of WebHDFS
hdfs_user	String	The user to access HDFS
root_directory	String	The root path of HDFS filesystem to store artifacts

A complete configuration example of `HDFSBlobManager` in `project.yaml`.

```
blob:
  blob_manager_class: ai_flow_plugins.blob_manager_plugins.hdfs_blob_manager.
↪HDFSBlobManager
  blob_manager_config:
    hdfs_url: http://hadoop-dfs:50070
    hdfs_user: hdfs
    root_directory: /tmp
```

S3BlobManager

// TODO

6.1.2 Using Blob Manager in a Workflow

The `Blob Manager` is not only be used by the AIFlow framework, users can also upload or download files with the `Blob Manager` if it has been configured in `project.yaml`. E.g.

```
from ai_flow.context.project_context import current_project_config
from ai_flow.workflow.workflow import WorkflowPropertyKeys
from ai_flow.plugin_interface.blob_manager_interface import BlobConfig,␣
↪BlobManagerFactory

blob_config = BlobConfig(current_project_config().get(WorkflowPropertyKeys.BLOB))
blob_manager = BlobManagerFactory.create_blob_manager(blob_config.blob_manager_
↪class(),
                                                      blob_config.blob_manager_
↪config())
blob_manager.upload(local_file_path='/tmp/file')
```

6.1.3 Customizing Blob Manager

You can also implement your own `Blob Manager` if the built-in ones do not meet your requirements. To create a blob manager plugin, one needs to implement a subclass of `ai_flow.plugin_interface.blob_manager_interface.BlobManager` to upload and download artifacts. To take configurations upon construction, the subclass should have a `__init__(self, config: Dict)` method. The configurations can be added when someone setup AIFlow to use the custom blob manager.

CHAPTER

SEVEN

HOW TOS

Setting up the sandbox in the *Quick Start* section was easy; building a production-grade environment requires a bit more work!

These how-to guides will step you through workflow development and setting up the AIFlow environment.

7.1 Set up MySQL as Backend

Both *AIFlow* and *Notification Server* support MySQL as backend during deployment. By default, AIFlow and Notification Server use SQLite, which is intended for development purposes only. This document will show you how to set up MySQL as backend.

7.1.1 Installing MySQL Client

To interacte with MySQL database, you need to install mysqlclient which is a MySQL database connector for Python.

Preparation

You need ensure that you have MySQL client libraries installed. You can check if you have installed locally by following command:

```
mysql_config --version
```

AIFlow is tested with **MySQL 5.7+**, if you are getting a lower version or `mysql_config: command not found` error, please following below commands to install MySQL client, otherwise you can skip this section.

macOS(Homebrew)

```
brew install mysql-client
echo 'export PATH="/usr/local/opt/mysql-client/bin:$PATH"' >> ~/.bash_profile
export PATH="/usr/local/opt/mysql-client/bin:$PATH"
```

Linux

```
sudo apt-get install python3-dev default-libmysqlclient-dev build-essential # Debian /
↪ Ubuntu
sudo yum install python3-devel mysql-devel # Red Hat / CentOS
```

Installing from PyPI

Now you can install mysqlclient with following command:

```
pip install 'ai-flow-nightly[mysql]'
```

7.1.2 Initializing Database

You need to create a database and a database user that AIFlow will use to access this database. In the example below, a database `aiflow` and user with username `admin` with password `admin` will be created

```
CREATE DATABASE aiflow CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
CREATE USER 'admin' IDENTIFIED BY 'admin';
GRANT ALL PRIVILEGES ON aiflow.* TO 'admin';
```

Note: The database must use a UTF-8 character set

After initializing database, you can create tables for AIFlow or Notification Server with command-line.

AIFlow

```
aiflow db init
```

Notification Server

```
notification db init
```

7.1.3 Configuring

Now you can modify the configurations about database connection to your mysql connection string of the following format

```
mysql+mysqldb://<user>:<password>@<host>[:<port>]/<dbname>
```

- For AIFLow you need to set `db_uri` to your mysql connection string and `db_type` to `MYSQL` in `aiflow_server.yaml`.
- For Notification Server you need to modify `db_uri` to your mysql connection string in `notification_server.yaml`.

CHAPTER

EIGHT

CLI

8.1 AIFlow

8.1.1 Command Line Interface

AIFlow has a very rich command-line interface that supports many types of operations on a Workflow, starting services and testing.

Content

- Positional Arguments
- Sub-commands:
  - *config*
  - *db*
  - *namespace*
  - *server*
  - *workflow*
  - *workflow-execution*
  - *task-execution*
  - *workflow-schedule*
  - *workflow-trigger*
  - *webserver*
  - *version*

```
usage: aiflow [-h] COMMAND ...
```

8.1.2 Positional Arguments

GROUP_OR_COMMAND

Possible choices: config, db, namespace, server, task-execution, workflow, workflow-execution, workflow-schedule, workflow-trigger, version.

8.1.3 Sub-commands

config

Manages configuration.

```
aiflow config [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: get-value, init, list.

Sub-commands

get-value

Gets the option value of the configuration.

```
aiflow config get-value [-h] option
```

Positional Arguments

option

The option name of the configuration.

init

Initializes the default configuration.

```
aiflow config init [-h]
```

list

Lists all options of the configuration.

```
aiflow config list [-h] [--color {auto,off,on}]
```

Named Arguments

–color

Possible choices: auto, off, onDo emit colored output (default: auto).Default: “auto”.

db

Database operations

```
aiflow db [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: downgrade, init, reset, upgrade.

Sub-commands

downgrade

Downgrades the metadata database to the version.

```
aiflow db downgrade [-h] [-v VERSION]
```

Named Arguments

-v, –version

The version corresponding to the database. Default: “heads”.

init

Initializes the metadata database.

```
aiflow db init [-h]
```

reset

Burns down and rebuild the metadata database.

```
aiflow db reset [-h] [-y]
```

Named Arguments

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

upgrade

Upgrades the metadata database to the version

```
aiflow db upgrade [-h] [-v VERSION]
```

Named Arguments

-v, –version

The version corresponding to the database.Default: “heads”.

namespace

Namespace related operations.

```
aiflow namespace [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: add, delete, list.

Sub-commands

add

Creates a namespace with specific name.

```
aiflow namespace add [-h] [--properties PROPERTIES] namespace_name
```

Positional Arguments

namespace_name

The name of the namespace.

Named Arguments

–properties

Properties of namespace, which is a string in json format.

delete

Deletes a namespace with specific name.

```
aiflow namespace delete [-h] [-y] namespace_name
```

Positional Arguments

namespace_name

The name of the namespace.

Named Arguments

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

list

Lists all the namespaces.

```
aiflow namespace list [-h] [-o table, json, yaml]
```

Named Arguments

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

server

AIFlow server operations.

```
aiflow server [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: start, stop.

Sub-commands

start

Starts the AIFlow server.

```
aiflow server start [-h] [-d]
```

Named Arguments

-d, –daemon

Daemonizes instead of running in the foreground.

stop

Stops the AIFlow server.

```
aiflow server stop [-h]
```

workflow

Workflow related operations.

```
aiflow workflow [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: delete, list, disable, enable, show, upload.

Sub-commands

delete

Deletes all DB records related to the specified workflow.

```
aiflow workflow delete [-h] [-n NAMESPACE] [-y] workflow_name
```

Positional Arguments

workflow_name

The name of the workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-y, –yes

Do not prompt to confirm reset. Use with care!Default: False.

list

Lists all the workflows.

```
aiflow workflow list [-h] [-n NAMESPACE] [-o table, json, yaml]
```

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

disable

Disables the workflow so that no more executions would be scheduled.

```
aiflow workflow disable [-h] [-n NAMESPACE] workflow_name
```

Positional Arguments

workflow_name

The name of the workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

enable

Enables the workflow which is disabled before.

```
aiflow workflow enable [-h] [-n NAMESPACE] workflow_name
```

Positional Arguments

workflow_name

The name of the workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

show

Shows the details of the workflow by workflow name.

```
aiflow workflow show [-h] [-n NAMESPACE] [-o table, json, yaml] workflow_name
```

Positional Arguments

workflow_name

The name of the workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

upload

Upload the workflow to the server along with artifacts.

```
aiflow workflow upload [-h] [-f FILES] file_path
```

Positional Arguments

file_path

The path of the workflow file

Named Arguments

-f, –files

Comma separated paths of files that would be uploaded along with the workflow.

workflow-execution

Workflow execution related operations.

```
aiflow workflow-execution [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: delete, list, show, start, stop, stop-all.

Sub-commands

delete

Deletes the workflow execution by execution id.

```
aiflow workflow-execution delete [-h] [-y] workflow_execution_id
```

Positional Arguments

workflow_execution_id

The id of the workflow execution.

Named Arguments

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

list

Lists all workflow executions of the workflow.

```
aiflow workflow-execution list [-h] [-n NAMESPACE] [-o table, json, yaml] workflow_
↪name
```

Positional Arguments

workflow_name

The name of workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

show

Shows the details of the workflow execution by execution id.

```
aiflow workflow-execution show [-h] [-o table, json, yaml] workflow_execution_id
```

Positional Arguments

workflow_execution_id

The id of the workflow execution

Named Arguments

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

start

Starts a new execution of the workflow.

```
aiflow workflow-execution start [-h] [-n NAMESPACE] workflow_name
```

Positional Arguments

workflow_name

The name of workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

stop

Stops the workflow execution by execution id.

```
aiflow workflow-execution stop [-h] workflow_execution_id
```

Positional Arguments

workflow_execution_id

The id of the workflow execution.

stop-all

Stops all workflow executions of the workflow.

```
aiflow workflow-execution stop-all [-h] [-n NAMESPACE] [-y] workflow_name
```

Positional Arguments

workflow_name

The name of workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

task-execution

Task execution related operations.

```
aiflow task-execution [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: list, show, start, stop.

Sub-commands

list

Lists all task executions of the workflow execution.

```
aiflow task-execution list [-h] [-o table, json, yaml] workflow_execution_id
```

Positional Arguments

workflow_execution_id

The id of the workflow execution.

Named Arguments

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

show

Shows the details of the task execution by execution id.

```
aiflow task-execution show [-h] [-o table, json, yaml] task_execution_id
```

Positional Arguments

task_execution_id

The id of the task execution.

Named Arguments

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

start

Starts a new execution of the task of the workflow execution.

```
aiflow task-execution start [-h] workflow_execution_id task_name
```

Positional Arguments

workflow_execution_id

The id of the workflow execution.

task_name

The name of the task.

stop

Stops the task execution by execution id.

```
aiflow task-execution stop [-h] workflow_execution_id task_name
```

Positional Arguments

workflow_execution_id

The id of the workflow execution.

task_name

The name of the task.

workflow-schedule

Manages the periodic schedules of the workflow.

```
aiflow workflow-schedule [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: add, delete, delete-all, list, pause, resume, show.

Sub-commands

add

Creates a new schedule for workflow.

```
aiflow workflow-schedule add [-h] [-n NAMESPACE] workflow_name expression
```

Positional Arguments

workflow_name

The name of workflow.

expression

The expression of the workflow schedule.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

delete

Deletes the workflow schedule by id.

```
aiflow workflow-schedule delete [-h] [-y] workflow_schedule_id
```

Positional Arguments

workflow_schedule_id

The id of the workflow schedule.

Named Arguments

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

delete-all

Deletes all schedules of the workflow.

```
aiflow workflow-schedule delete-all [-h] [-n NAMESPACE] [-y] workflow_name
```

Positional Arguments

workflow_name

The name of workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

list

Lists all schedules of the workflow.

```
aiflow workflow-schedule list [-h] [-n NAMESPACE] [-o table, json, yaml] workflow_name
```

Positional Arguments

workflow_name

The name of workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

pause

Pauses the schedule and the workflow would not periodically execute anymore.

```
aiflow workflow-schedule pause [-h] workflow_schedule_id
```

Positional Arguments

workflow_schedule_id

The id of the workflow schedule.

resume

Resumes the schedule which is paused before.

```
aiflow workflow-schedule resume [-h] workflow_schedule_id
```

Positional Arguments

workflow_schedule_id

The id of the workflow schedule.

show

Shows the details of the workflow schedule by id.

```
aiflow workflow-schedule show [-h] [-o table, json, yaml] workflow_schedule_id
```

Positional Arguments

workflow_schedule_id

The id of the workflow schedule.

Named Arguments

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

workflow-trigger

Manages the event triggers of the workflow.

```
aiflow workflow-trigger [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: delete, delete-all, list, pause, resume, show.

Sub-commands

delete

Deletes the workflow event trigger by id.

```
aiflow workflow-trigger delete [-h] [-y] workflow_trigger_id
```

Positional Arguments

workflow_trigger_id

The id of the workflow trigger.

Named Arguments

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

delete-all

Deletes all event triggers of the workflow.

```
aiflow workflow-trigger delete-all [-h] [-n NAMESPACE] [-y] workflow_name
```

Positional Arguments

workflow_name

The name of workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

list

Lists all event triggers of the workflow.

```
aiflow workflow-trigger list [-h] [-n NAMESPACE] [-o table, json, yaml] workflow_name
```

Positional Arguments

workflow_name

The name of workflow.

Named Arguments

-n, –namespace

Namespace that contains the workflow.

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

pause

Pauses the event trigger by id.

```
aiflow workflow-trigger pause [-h] workflow_trigger_id
```

Positional Arguments

workflow_trigger_id

The id of the workflow trigger.

resume

Resumes the event trigger by id.

```
aiflow workflow-trigger resume [-h] workflow_trigger_id
```

Positional Arguments

workflow_trigger_id

The id of the workflow trigger.

show

Shows the details of the workflow event trigger by id.

```
aiflow workflow-trigger show [-h] [-o table, json, yaml] workflow_trigger_id
```

Positional Arguments

workflow_trigger_id

The id of the workflow trigger.

Named Arguments

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

webserver

AIFlow Webserver operations.

```
aiflow webserver [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: start, stop.

Sub-commands

start

Starts the AIFlow Webserver.

```
aiflow webserver start [-h] [-d]
```

Named Arguments

-d, –daemon

Daemonizes instead of running in the foreground.

stop

Stops the AIFlow Webserver

```
aiflow webserver stop [-h]
```

version

Shows the version.

```
aiflow version [-h]
```

8.2 Notification

8.2.1 Command Line Interface

Notification has a very rich command-line interface that supports many types of operations on Events, starting services and testing.

Content

- Positional Arguments
- Sub-commands:
  - *server*
  - *event*
  - *config*
  - *db*
  - *version*

notification

```
usage: notification [-h] COMMAND ...
```

Positional Arguments

GROUP_OR_COMMAND

Possible choices: server, event, config, db, version.

Sub-commands

server

Notification server operations.

```
notification server [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: start, stop.

Sub-commands

start

Starts the notification server.

```
notification server start [-h] [-d]
```

Named Arguments

-d, –daemon

Daemonizes instead of running in the foreground.

stop

Stops the notification server.

```
notification server stop [-h]
```

event

Manages events.

```
notification event [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: count, list, listen, send.

Sub-commands

count

Counts events.

```
notification event count [-h] [--begin-offset BEGIN_OFFSET] [--begin-time BEGIN_TIME]␣
↪[-n NAMESPACE] [--sender SENDER] [-s SERVER_URI] key
```

Positional Arguments

key

Key of the event.

Named Arguments

-s, –server-uri

The uri of notification server.

-n, –namespace

Namespace of the event. If not set, all namespaces would be handled.

–begin-offset

Begin offset of the event. Defaults to 0

–begin-time

Begin datetime of the event, formatted in ISO 8601.

–sender

Sender of the event.

list

Lists events.

```
notification event list [-h] [--begin-offset BEGIN_OFFSET] [--begin-time BEGIN_TIME]␣
↪[-n NAMESPACE] [-o table, json, yaml] [--sender SENDER] [-s SERVER_URI] key
```

Positional Arguments

key

Key of the event.

Named Arguments

-s, –server-uri

The uri of notification server.

-n, –namespace

Namespace of the event. If not set, all namespaces would be handled.

–begin-offset

Begin offset of the event. Defaults to 0

–begin-time

Begin datetime of the event, formatted in ISO 8601.

–sender

Sender of the event.

-o, –output

Possible choices: table, json, yaml, plain.Output format. Allowed values: json, yaml, plain, table (default: table).Default: “table”.

listen

Listens events

```
notification event listen [-h] [--begin-offset BEGIN_OFFSET] [--begin-time BEGIN_
↪TIME] [-n NAMESPACE] [-s SERVER_URI] key
```

Positional Arguments

key

Key of the event.

Named Arguments

-s, –server-uri

The uri of notification server.

-n, –namespace

Namespace of the event. If not set, all namespaces would be handled.

–begin-offset

Begin offset of the event. Defaults to 0

–begin-time

Begin datetime of the event to listen, formatted in ISO 8601. Default: `datetime.now().isoformat()`.

send

Sends an event.

```
notification event send [-h] [--context CONTEXT] [-n NAMESPACE] [--sender SENDER] [-s␣
↪SERVER_URI] key value
```

Positional Arguments

key

Key of the event.

value

Value of the event.

Named Arguments

-s, –server-uri

The uri of notification server.

-n, –namespace

Namespace of the event. If not set, all namespaces would be handled.

–context

Context of the event.

–sender

Sender of the event.

config

Manages configuration.

```
notification config [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: get-value, init, list.

Sub-commands

get-value

Gets the option value of the configuration.

```
notification config get-value [-h] option
```

Positional Arguments

option

The option name of the configuration.

init

Initializes the default configuration.

```
notification config init [-h]
```

list

Lists all options of the configuration.

```
notification config list [-h] [--color {auto,off,on}]
```

Named Arguments

–color

Possible choices: auto, off, onDo emit colored output (default: auto).Default: “auto”.

db

Database operations

```
notification db [-h] COMMAND ...
```

Positional Arguments

COMMAND

Possible choices: downgrade, init, reset, upgrade.

Sub-commands

downgrade

Downgrades the metadata database to the version.

```
notification db downgrade [-h] [-v VERSION]
```

Named Arguments

-v, –version

The version corresponding to the database. Default: “heads”.

init

Initializes the metadata database.

```
notification db init [-h]
```

reset

Burns down and rebuild the metadata database.

```
notification db reset [-h] [-y]
```

Named Arguments

-y, –yes

Do not prompt to confirm reset. Use with care! Default: False.

upgrade

Upgrades the metadata database to the version

```
notification db upgrade [-h] [-v VERSION]
```

Named Arguments

-v, –version

The version corresponding to the database.Default: “heads”.

version

Shows the version.

```
notification version [-h]
```

CHAPTER

NINE

API

9.1 Python

9.1.1 ai_flow package

Subpackages

ai_flow.model package

Subpackages

Submodules

ai_flow.model.action module

class ai_flow.model.action.**TaskAction**(*value*)
Bases: str, enum.Enum

Enumeration of execution commands for scheduled tasks. START: Start a task instance. RESTART: Stop the current task instance and start a new task instance. STOP: Stop a task instance.

RESTART = 'RESTART'

START = 'START'

STOP = 'STOP'

ai_flow.model.condition module

class ai_flow.model.condition.**Condition**(*expect_event_keys: List[str]*)
Bases: object

Conditions that trigger scheduling.

Parameters **expect_event_keys** – The keys of events that this condition depends on.

abstract is_met(*event: notification_service.model.event.Event*, *context:* ai_flow.model.context.Context) → bool
Determine whether the condition is met. :param event: The currently processed event. :param context: The context in which the condition is executed. :return True:The condition is met. False: The condition is not met.

ai_flow.model.context module

class ai_flow.model.context.**Context**
Bases: object

The context in which custom logic is executed.

get_state(*state_descriptor:* ai_flow.model.state.StateDescriptor) → *ai_flow.model.state.State*
Get the State object. :param state_descriptor: Description of the State object. :return The State object.

get_task_status(*task_name*) → *ai_flow.model.status.TaskStatus*
Get the task status by task name. :param task_name: The name of the task.

ai_flow.model.execution_type module

class ai_flow.model.execution_type.**ExecutionType**(*value*)
Bases: str, enum.Enum

Enumeration of execution of workflow and task. MANUAL: Manually trigger execution. EVENT: Event triggered execution. PERIODIC: Periodic triggered execution.

EVENT = 'EVENT'

MANUAL = 'MANUAL'

PERIODIC = 'PERIODIC'

ai_flow.model.operator module

class ai_flow.model.operator.**AIFlowOperator**(*task_name: str*, ***kwargs*)
Bases: *ai_flow.model.operator.Operator*

AIFlowOperator is a template that defines a task, it defines AIFlow's native Operator interface. To derive this class, you are expected to override the constructor as well as abstract methods.

Parameters

- **name** – The operator's name.
- **kwargs** – Operator's extended parameters.

await_termination(*context:* ai_flow.model.context.Context, *timeout: Optional[int] = None*)
Wait for a task instance to finish. :param context: The context in which the operator is executed. :param timeout: If timeout is None, wait until the task ends.

If timeout is not None, wait for the task to end or the time exceeds timeout(seconds).

get_metrics(*context:* ai_flow.model.context.Context) → Dict
Get the metrics of a task instance.

abstract start(*context:* ai_flow.model.context.Context)
Start a task instance.

stop(*context:* ai_flow.model.context.Context)
Stop a task instance.

class ai_flow.model.operator.**Operator**(*name: str*, ***kwargs*)
Bases: object

Operator is a template that defines a task. It is the abstract base class for all operators. Since operators create objects that become tasks in the Workflow.To derive this class, you are expected to override the constructor

method. This class is abstract and shouldn't be instantiated. Instantiating a class derived from this one results in the creation of a task object, which ultimately becomes a task in Workflow objects.

Parameters

- **name** – The operator's name.
- **kwargs** – Operator's extended parameters.

action_on_condition(*action:* ai_flow.model.action.TaskAction, *condition:* ai_flow.model.condition.Condition)
Schedule the task based on a specified condition. :param action: The action for scheduling the task. :param condition: The condition for scheduling the task to depend on.

action_on_event_received(*action:* ai_flow.model.action.TaskAction, *event_key: str*)
When the specified event is received, the task is scheduled. :param action: The action for scheduling the task. :param event_key: The event for scheduling the task to depend on.

action_on_task_status(*action:* ai_flow.model.action.TaskAction, *upstream_task_status_dict:* *Dict[*ai_flow.model.operator.Operator, ai_flow.model.status.TaskStatus*]*)
Schedule the task based on the status of upstream tasks. :param action: The action for scheduling the task. :param upstream_task_status_dict: The upstream task status for scheduling the task to depend on.

start_after(*tasks: Union[*ai_flow.model.operator.Operator, *List[*ai_flow.model.operator.Operator*]]*)
Start the task after upstream tasks succeed. :param tasks: The upstream tasks.

class ai_flow.model.operator.**OperatorConfigItem**
Bases: object

The Operator's config items. PERIODIC_EXPRESSION: The expression for the periodic task.

PERIODIC_EXPRESSION = 'periodic_expression'

ai_flow.model.rule module

class ai_flow.model.rule.**TaskRule**(*condition:* ai_flow.model.condition.Condition, *action:* ai_flow.model.action.TaskAction)
Bases: object

Rules that trigger task scheduling.

Parameters

- **condition** – Trigger condition of the rule.
- **action** – The execution commands for scheduled tasks.

trigger(*event: notification_service.model.event.Event*, *context:* ai_flow.model.context.Context) → Optional[*ai_flow.model.action.TaskAction*]
Determine whether to trigger task scheduling behavior. :param event: The currently processed event. :param context: The context in which the rule is executed. :return None: Does not trigger task scheduling behavior.

Not None: Execution command for scheduling the task.

class ai_flow.model.rule.**WorkflowRule**(*condition:* ai_flow.model.condition.Condition)
Bases: object

Rules that trigger workflow scheduling.

Parameters **condition** – Trigger condition of the rule.

trigger(*event: notification_service.model.event.Event*, *context:* ai_flow.model.context.Context) → bool

Determine whether to trigger workflow running. :param event: The currently processed event. :param context: The context in which the rule is executed. :return True:Start a WorkflowExecution. False: Do not start a WorkflowExecution.

ai_flow.model.state module

class ai_flow.model.state.**State**

Bases: object

User-defined state

clear()

Clean up user-defined state

class ai_flow.model.state.**StateDescriptor**(*name*)

Bases: object

User-defined state description

class ai_flow.model.state.**StateType**

Bases: object

VALUE = 'value'

class ai_flow.model.state.**ValueState**

Bases: *ai_flow.model.state.State*

Single-valued user-defined state

update(*state*)

Update the single-valued user-defined state's value

value() → object

Get the single-valued user-defined state's value

class ai_flow.model.state.**ValueStateDescriptor**(*name*)

Bases: *ai_flow.model.state.StateDescriptor*

Single-valued user-defined state description

ai_flow.model.status module

class ai_flow.model.status.**TaskStatus**(*value*)

Bases: str, enum.Enum

Enumeration of TaskExecution's status. INIT: The initial status of TaskExecution. QUEUED: The TaskExecution has been assigned to an executor. RESTARTING: The TaskExecution was requested to restart when it was running RUNNING: The TaskExecution is running. SUCCESS: The TaskExecution finished running without errors. FAILED: The TaskExecution had errors during execution and failed to run. KILLING: The TaskExecution was externally requested to shut down when it was running. KILLED: The TaskExecution was externally shut down. RETRYING: The TaskExecution failed, but has retry attempts left and will be rescheduled.

FAILED = 'FAILED'

INIT = 'INIT'

QUEUED = 'QUEUED'

RETRYING = 'RETRYING'

RUNNING = 'RUNNING'

STOPPED = 'STOPPED'

STOPPING = 'STOPPING'

SUCCESS = 'SUCCESS'

class ai_flow.model.status.**WorkflowStatus**(*value*)

Bases: str, enum.Enum

Enumeration of WorkflowExecution's status. INIT: The initial status of WorkflowExecution. RUNNING: The WorkflowExecution is running. SUCCESS: The WorkflowExecution finished running without errors. FAILED: The WorkflowExecution had errors during execution and failed to run. STOPPED: The WorkflowExecution has been stopped.

FAILED = 'FAILED'

INIT = 'INIT'

RUNNING = 'RUNNING'

STOPPED = 'STOPPED'

SUCCESS = 'SUCCESS'

ai_flow.model.task_execution module

class ai_flow.model.task_execution.**TaskExecution**(*workflow_execution_id: int*, *task_name: str*, *sequence_number: int*, *execution_type: ai_flow.model.execution_type.ExecutionType*, *begin_date: Optional[datetime.datetime] = None*, *end_date: Optional[datetime.datetime] = None*, *status: ai_flow.model.status.TaskStatus = <TaskStatus.INIT: 'INIT'>*, *id: Optional[int] = None*)

Bases: object

TaskExecution describes an instance of a task. It can be created by the scheduler.

Parameters

- **workflow_execution_id** – TaskExecution belongs to the unique identifier of WorkflowExecution.
- **task_name** – The name of the task it belongs to.
- **sequence_number** – A task in a WorkflowExecution can be run multiple times, it indicates how many times this task is run.
- **execution_type** – The type that triggers TaskExecution.
- **begin_date** – The time TaskExecution started executing.
- **end_date** – The time TaskExecution ends execution.
- **status** – TaskExecution's current status.
- **id** – Unique ID of TaskExecution.

class ai_flow.model.task_execution.**TaskExecutionKey**(*workflow_execution_id*, *task_name*, *seq_num*)

Bases: object

ai_flow.model.workflow module

class ai_flow.model.workflow.**Workflow**(*name: str*, *namespace: str = 'default'*, ***kwargs*)

Bases: object

Workflow is a collection of tasks and trigger rules. A Workflow can be scheduled by events, manual or schedule. For each execution, the workflow needs to run its individual tasks when their triggering rules are met. Workflows essentially act as namespaces for tasks. A task_id can only be added once to a Workflow.

Parameters **name** – The name of the workflow.

action_on_condition(*task_name: str*, *action:* ai_flow.model.action.TaskAction, *condition:* ai_flow.model.condition.Condition)

action_on_event_received(*task_name: str*, *event_key: str*, *action:* ai_flow.model.action.TaskAction)

action_on_task_status(*task_name: str*, *action:* ai_flow.model.action.TaskAction, *upstream_task_status_dict: Dict[*ai_flow.model.operator.Operator, ai_flow.model.status.TaskStatus*]*)

class ai_flow.model.workflow.**WorkflowContextManager**

Bases: object

Workflow context manager is used to keep the current Workflow when Workflow is used as ContextManager. You can use Workflow as context: .. code-block:: python

with Workflow(name = 'workflow'

) as workflow: ...

If you do this the context stores the Workflow and whenever new task is created, it will use such Workflow as the parent Workflow.

classmethod get_current_workflow() → Optional[*ai_flow.model.workflow.Workflow*]

classmethod pop_context_managed_workflow() → Optional[*ai_flow.model.workflow.Workflow*]

classmethod push_context_managed_workflow(*workflow:* ai_flow.model.workflow.Workflow)

ai_flow.model.workflow_execution module

class ai_flow.model.workflow_execution.**WorkflowExecution**(*workflow_id*, *execution_type:* ai_flow.model.execution_type.ExecutionType, *begin_date: datetime.datetime*, *end_date: datetime.datetime*, *status:* ai_flow.model.status.WorkflowStatus, *id: Optional[int] = None*)

Bases: object

WorkflowExecution describes an instance of a Workflow. It can be created by the scheduler.

Parameters

- **workflow_id** – WorkflowExecution belongs to the unique identifier of Workflow.
- **execution_type** – The type that triggers WorkflowExecution.
- **begin_date** – The time WorkflowExecution started executing.
- **end_date** – The time WorkflowExecution ends execution.
- **status** – WorkflowExecution's current status.
- **id** – Unique ID of WorkflowExecution.

ai_flow.notification package

Submodules

ai_flow.notification.notification_client module

class ai_flow.notification.notification_client.**AIFlowNotificationClient**(*server_uri: str*)

Bases: object

close()

register_listener(*listener_processor: notification_service.client.notification_client.ListenerProcessor*, *event_keys: Optional[List[str]] = None*, *begin_time: Optional[datetime.datetime] = None*) → notification_service.client.notification_client.ListenerRegistrationId

send_event(*key: str*, *value: Optional[str] = None*)

Send event to current workflow execution. This function can only be used in AIFlow Operator runtime. It will retrieve the workflow execution info from runtime context and set to context of the event.

Parameters

- **key** – the key of the event.
- **value** – optional, the value of the event.

unregister_listener(*registration_id: notification_service.client.notification_client.ListenerRegistrationId*)

ai_flow.operators package

Subpackages

ai_flow.operators.flink package

Submodules

ai_flow.operators.flink.flink_operator module

class ai_flow.operators.flink.flink_operator.**FlinkOperator**(*name: str, application: str, target: Optional[str] = None, application_args: Optional[List[Any]] = None, executable_path: Optional[str] = None, application_mode: bool = False, stop_with_savepoint: bool = False, kubernetes_cluster_id: Optional[str] = None, command_options: Optional[str] = None, jobs_info_poll_interval: int = 1, **kwargs*)

Bases: *ai_flow.model.operator.AIFlowOperator*

FlinkOperator is used to submit Flink job with flink command line.

Parameters

- **name** – The name of the operator.
- **application** – The application file to be submitted, like app jar, python file.
- **target** – The deployment target for the given application, which is equivalent to the "execution.target" config option.
- **application_args** – Args of the application.
- **executable_path** – The path of flink command.
- **application_mode** – Whether runs an application in Application Mode.
- **stop_with_savepoint** – Whether stops the flink job with a savepoint.
- **kubernetes_cluster_id** – Cluster id when submit flink job to kubernetes.
- **command_options** – The options that passes to command-line, e.g. -D, –class and –classpath.
- **jobs_info_poll_interval** – Seconds to wait between polls of job status in application mode (Default: 1)
- **name** – The operator's name.
- **kwargs** – Operator's extended parameters.

await_termination(*context:* ai_flow.model.context.Context, *timeout: Optional[int] = None*)

Wait for a task instance to finish. :param context: The context in which the operator is executed. :param timeout: If timeout is None, wait until the task ends.

If timeout is not None, wait for the task to end or the time exceeds timeout(seconds).

start(*context:* ai_flow.model.context.Context)

Start a task instance.

stop(*context:* ai_flow.model.context.Context)
Stop a task instance.

ai_flow.operators.spark package

Submodules

ai_flow.operators.spark.spark_sql module

class ai_flow.operators.spark.spark_sql.**SparkSqlOperator**(*name: str*, *sql: str*, *master: str = 'yarn'*, *application_name: Optional[str] = None*, *executable_path: Optional[str] = None*, ***kwargs*)

Bases: *ai_flow.model.operator.AIFlowOperator*

SparkSqlOperator only supports client mode for now.

Parameters

- **name** – The operator's name.
- **kwargs** – Operator's extended parameters.

await_termination(*context:* ai_flow.model.context.Context, *timeout: Optional[int] = None*)
Wait for a task instance to finish. :param context: The context in which the operator is executed. :param timeout: If timeout is None, wait until the task ends.

If timeout is not None, wait for the task to end or the time exceeds timeout(seconds).

start(*context:* ai_flow.model.context.Context)
Start a task instance.

stop(*context:* ai_flow.model.context.Context)
Stop a task instance.

ai_flow.operators.spark.spark_submit module

class ai_flow.operators.spark.spark_submit.**SparkSubmitOperator**(*name: str*, *application: str*, *application_args: Optional[List[Any]] = None*, *executable_path: Optional[str] = None*, *master: str = 'yarn'*, *deploy_mode: str = 'client'*, *application_name: Optional[str] = None*, *submit_options: Optional[str] = None*, *k8s_namespace: Optional[str] = None*, *env_vars: Optional[Dict[str, Any]] = None*, ***kwargs*)

Bases: *ai_flow.model.operator.AIFlowOperator*

SparkSubmitOperator is used to submit spark job with spark-submit command line.

Parameters

- **name** – The name of the operator.
- **application** – The application file to be submitted, like app jar, python file or R file.
- **application_args** – Args of the application.
- **executable_path** – The path of spark-submit command.
- **master** – spark://host:port, yarn, mesos://host:port, k8s://https://host:port, or local.
- **deploy_mode** – Launch the program in client(by default) mode or cluster mode.
- **application_name** – The name of spark application.
- **submit_options** – The options that passes to command-line, e.g. –conf, –class and –files
- **k8s_namespace** – The namespace of k8s, when submit application to k8s, it should be passed.
- **env_vars** – Environment variables for spark-submit. It supports yarn and k8s mode too.
- **status_poll_interval** – Seconds to wait between polls of driver status in cluster mode (Default: 1)
- **name** – The operator's name.
- **kwargs** – Operator's extended parameters.

await_termination(*context:* ai_flow.model.context.Context, *timeout: Optional[int] = None*)
Wait for a task instance to finish. :param context: The context in which the operator is executed. :param timeout: If timeout is None, wait until the task ends.

If timeout is not None, wait for the task to end or the time exceeds timeout(seconds).

start(*context:* ai_flow.model.context.Context)
Start a task instance.

stop(*context:* ai_flow.model.context.Context)
Stop a task instance.

Submodules

ai_flow.operators.bash module

class ai_flow.operators.bash.**BashOperator**(*name: str*, *bash_command: str*, ***kwargs*)
Bases: *ai_flow.model.operator.AIFlowOperator*

Parameters

- **name** – The operator's name.
- **kwargs** – Operator's extended parameters.

await_termination(*context:* ai_flow.model.context.Context, *timeout: Optional[int] = None*)
Wait for a task instance to finish. :param context: The context in which the operator is executed. :param timeout: If timeout is None, wait until the task ends.

If timeout is not None, wait for the task to end or the time exceeds timeout(seconds).

start(*context:* ai_flow.model.context.Context)
Start a task instance.

stop(*context:* ai_flow.model.context.Context)
Stop a task instance.

ai_flow.operators.python module

class ai_flow.operators.python.**PythonOperator**(*name: str*, *python_callable: Callable*, *op_args: Optional[List] = None*, *op_kwargs: Optional[Dict] = None*, ***kwargs*)
Bases: *ai_flow.model.operator.AIFlowOperator*

Parameters

- **name** – The operator's name.
- **kwargs** – Operator's extended parameters.

start(*context:* ai_flow.model.context.Context)
Start a task instance.

ai_flow.ops package

Submodules

ai_flow.ops.namespace_ops module

ai_flow.ops.namespace_ops.**add_namespace**(*name: str*, *properties: dict*) → ai_flow.metadata.namespace.NamespaceMeta

Creates a new namespace in metadata.

Parameters

- **name** – The name of namespace to be added.
- **properties** – The properties of namespace.

Returns The NamespaceMeta instance just added.

ai_flow.ops.namespace_ops.**delete_namespace**(*name: str*)

Deletes the namespace from metadata.

Parameters **name** – The name of namespace.

Raises AIFlowException if failed to delete namespace.

ai_flow.ops.namespace_ops.**get_namespace**(*name: str*) → Optional[ai_flow.metadata.namespace.NamespaceMeta]

Retrieves the namespace from metadata.

Parameters **name** – The name of namespace.

Returns The NamespaceMeta instance, return None if no namespace found.

ai_flow.ops.namespace_ops.**list_namespace**(*limit: Optional[int] = None*, *offset: Optional[int] = None*) → Optional[List[ai_flow.metadata.namespace.NamespaceMeta]]

Retrieves the list of namespaces from metadata.

Parameters

- **limit** – The maximum records to be listed.
- **offset** – The offset to start to list.

Returns The NamespaceMeta list, return None if no namespace found.

ai_flow.ops.namespace_ops.**update_namespace**(*name: str*, *properties: dict*) → Optional[ai_flow.metadata.namespace.NamespaceMeta]

Updates the properties of the namespace.

Parameters

- **name** – The name of namespace to be updated.
- **properties** – The properties of namespace.

Returns The NamespaceMeta instance just updated, return None if no namespace found.

ai_flow.ops.task_execution_ops module

ai_flow.ops.task_execution_ops.**get_task_execution**(*task_execution_id: int*) → ai_flow.metadata.task_execution.TaskExecutionMeta

Retrieves the task execution from metadata.

Parameters **task_execution_id** – The id of the task execution.

Returns The TaskExecutionMeta instance, return None if no execution found.

ai_flow.ops.task_execution_ops.**list_task_executions**(*workflow_execution_id: int*, *limit: Optional[int] = None*, *offset: Optional[int] = None*) → Optional[List[ai_flow.metadata.task_execution.TaskExecutio

Retrieves the list of executions of the task of the workflow execution.

Parameters

- **workflow_execution_id** – The id of the workflow execution.
- **limit** – The maximum records to be listed.
- **offset** – The offset to start to list.

Returns The TaskExecutionMeta list, return None if no task execution found.

ai_flow.ops.task_execution_ops.**start_task_execution**(*workflow_execution_id: int*, *task_name: str*) → str

Start a new execution of the task.

Parameters

- **workflow_execution_id** – The workflow execution contains the task.
- **task_name** – The name of the task to be started.

Returns The TaskExecutionKey str.

Raises AIFlowException if failed to start task execution.

ai_flow.ops.task_execution_ops.**stop_task_execution**(*workflow_execution_id: int*, *task_name: str*)

Asynchronously stop the task execution.

Parameters

- **workflow_execution_id** – The workflow execution contains the task.
- **task_name** – The name of the task to be stopped.

Raises AIFlowException if failed to stop task execution.

ai_flow.ops.workflow_execution_ops module

ai_flow.ops.workflow_execution_ops.**delete_workflow_execution**(*workflow_execution_id: int*)

Deletes the workflow execution from metadata, note that the workflow execution to be deleted should be finished.

Parameters **workflow_execution_id** – The id of the workflow execution.

Raises AIFlowException if failed to delete the workflow execution.

ai_flow.ops.workflow_execution_ops.**get_workflow_execution**(*workflow_execution_id: int*) → Optional[ai_flow.metadata.workflow_execution.Wo

Retrieves the workflow execution from metadata.

Parameters **workflow_execution_id** – The id of the workflow execution.

Returns The WorkflowExecutionMeta instance, return None if no execution found.

ai_flow.ops.workflow_execution_ops.**list_workflow_executions**(*workflow_name: str*, *namespace: str = 'default'*, *limit: Optional[int] = None*, *offset: Optional[int] = None*) → Optional[List[ai_flow.metadata.workflow_execu

Retrieves the list of executions of the workflow.

Parameters

- **workflow_name** – The workflow to be listed.
- **namespace** – The namespace which contains the workflow.
- **limit** – The maximum records to be listed.
- **offset** – The offset to start to list.

Returns The WorkflowExecutionMeta list, return None if no workflow execution found.

ai_flow.ops.workflow_execution_ops.**start_workflow_execution**(*workflow_name: str*, *namespace: str = 'default'*) → int

Start a new execution of the workflow.

Parameters

- **workflow_name** – The workflow to be executed.
- **namespace** – The namespace which contains the workflow.

Returns Id of the workflow execution just started.

Raises AIFlowException if failed to start workflow execution.

ai_flow.ops.workflow_execution_ops.**stop_workflow_execution**(*workflow_execution_id: int*)

Asynchronously stop the execution of the workflow.

Parameters **workflow_execution_id** – The id of workflow execution to be stopped.

Raises AIFlowException if failed to stop the workflow execution.

ai_flow.ops.workflow_execution_ops.**stop_workflow_executions**(*workflow_name: str*, *namespace: str = 'default'*)

Asynchronously stop all executions of the workflow.

Parameters

- **workflow_name** – The workflow to be stopped.
- **namespace** – The namespace which contains the workflow.

Raises AIFlowException if failed to stop workflow executions.

ai_flow.ops.workflow_ops module

ai_flow.ops.workflow_ops.**delete_workflow**(*workflow_name: str*, *namespace: str = 'default'*)
Deletes the workflow from metadata, also its executions, schedules and triggers would be cascade deleted, however if not-finished workflow execution found, the deletion would be interrupted.

Parameters

- **workflow_name** – The name of the workflow.
- **namespace** – The namespace of the workflow.

Raises AIFlowException if failed to delete the workflow.

ai_flow.ops.workflow_ops.**disable_workflow**(*workflow_name: str*, *namespace: str = 'default'*)
Disables the workflow so that no more executions would be started, however, the existed executions are not effected.

Parameters

- **workflow_name** – The name of the workflow.
- **namespace** – The namespace of the workflow.

Raises AIFlowException if failed to disable workflow.

ai_flow.ops.workflow_ops.**enable_workflow**(*workflow_name: str*, *namespace: str = 'default'*)
Enables the workflow.

Parameters

- **workflow_name** – The name of the workflow.
- **namespace** – The namespace of the workflow.

Raises AIFlowException if failed to enable workflow.

ai_flow.ops.workflow_ops.**get_workflow**(*workflow_name: str*, *namespace: str = 'default'*) → Optional[ai_flow.metadata.workflow.WorkflowMeta]
Retrieves the workflow from metadata.

Parameters

- **workflow_name** – The name of the workflow.
- **namespace** – The namespace of the workflow.

Returns The WorkflowMeta instance, return None if no workflow found.

ai_flow.ops.workflow_ops.**list_workflows**(*namespace: str = 'default'*, *limit: Optional[int] = None*, *offset: Optional[int] = None*) → Optional[List[ai_flow.metadata.workflow.WorkflowMeta]]
Retrieves the list of workflow of the namespace from metadata.

Parameters

- **namespace** – The namespace of the workflow.
- **limit** – The maximum records to be listed.
- **offset** – The offset to start to list.

Returns The WorkflowMeta list, return None if no workflow found.

ai_flow.ops.workflow_ops.**upload_workflows**(*workflow_file_path: str*, *artifacts: Optional[List[str]] = None*) → List[ai_flow.metadata.workflow.WorkflowMeta]

Upload the workflow defined in *workflow_file_path* along with it's dependencies to AIFlow server.

Parameters

- **workflow_file_path** – The path of the workflow to be uploaded.
- **artifacts** – The artifacts that the workflow needed.

Returns The uploaded workflows.

ai_flow.ops.workflow_schedule_ops module

ai_flow.ops.workflow_schedule_ops.**add_workflow_schedule**(*expression: str*, *workflow_name: str*, *namespace: str = 'default'*) → ai_flow.metadata.workflow_schedule.WorkflowSch

Creates a new workflow schedule in metadata.

Parameters

- **expression** – The string express when the workflow execution is triggered. Two types of expression are supported here: cron and interval. cron_expression:

  cron@minute, hour, day of month, month, day of week See https://en.wikipedia.org/wiki/Cron for more information on the format accepted here.

  interval_expression: interval@days hours minutes seconds e.g. "interval@0 1 0 0" means running every 1 hour since now.

- **workflow_name** – The name of the workflow to be registered schedule.
- **namespace** – The namespace of the workflow.

Returns The WorkflowScheduleMeta instance just added.

ai_flow.ops.workflow_schedule_ops.**delete_workflow_schedule**(*schedule_id*)

Deletes the workflow schedule from metadata.

Parameters **schedule_id** – The id of the workflow schedule.

Raises AIFlowException if failed to delete the workflow schedule.

ai_flow.ops.workflow_schedule_ops.**delete_workflow_schedules**(*workflow_name: str*, *namespace: str = 'default'*)

Deletes all schedules of the workflow.

Parameters

- **workflow_name** – The name of the workflow.
- **namespace** – The namespace which contains the workflow.

Raises AIFlowException if failed to delete workflow schedules.

ai_flow.ops.workflow_schedule_ops.**get_workflow_schedule**(*schedule_id: int*) → Optional[ai_flow.metadata.workflow_schedule.Workfl

Retrieves the workflow schedule from metadata.

Parameters **schedule_id** – The id of the schedule.

Returns The WorkflowScheduleMeta instance, return None if no schedule found.

ai_flow.ops.workflow_schedule_ops.**list_workflow_schedules**(*workflow_name: str*, *namespace: str = 'default'*, *limit: Optional[int] = None*, *offset: Optional[int] = None*) → Optional[List[ai_flow.metadata.workflow_schedule

Retrieves the list of schedules of the workflow.

Parameters

- **workflow_name** – The workflow to be listed schedules.
- **namespace** – The namespace which contains the workflow.
- **limit** – The maximum records to be listed.
- **offset** – The offset to start to list.

Returns The WorkflowScheduleMeta list, return None if no workflow schedules found.

ai_flow.ops.workflow_schedule_ops.**pause_workflow_schedule**(*schedule_id: int*)

Pauses the workflow schedule.

Parameters **schedule_id** – The id of the workflow schedule.

Raises AIFlowException if failed to pause the workflow schedule.

ai_flow.ops.workflow_schedule_ops.**resume_workflow_schedule**(*schedule_id: int*)

Resumes the workflow schedule which is paused before.

Parameters **schedule_id** – The id of the workflow schedule.

Raises AIFlowException if failed to resume the workflow schedule.

ai_flow.ops.workflow_snapshot_ops module

ai_flow.ops.workflow_snapshot_ops.**delete_workflow_snapshot**(*snapshot_id: int*)

Deletes the workflow snapshot from metadata.

Parameters **snapshot_id** – The id of the workflow snapshot.

Raises AIFlowException if failed to delete the workflow snapshot.

ai_flow.ops.workflow_snapshot_ops.**delete_workflow_snapshots**(*workflow_name: str*, *namespace: str = 'default'*)

Deletes all snapshots of the workflow.

Parameters

- **workflow_name** – The name of the workflow.
- **namespace** – The namespace which contains the workflow.

Raises AIFlowException if failed to delete workflow snapshots.

ai_flow.ops.workflow_snapshot_ops.**get_workflow_snapshot**(*snapshot_id: int*) → Optional[ai_flow.metadata.workflow_snapshot.Workfl

Retrieves the workflow snapshot from metadata.

Parameters **snapshot_id** – The id of the snapshot.

Returns The WorkflowSnapshotMeta instance, return None if no snapshot found.

ai_flow.ops.workflow_snapshot_ops.**list_workflow_snapshots**(*workflow_name: str*, *namespace: str = 'default'*, *limit: Optional[int] = None*, *offset: Optional[int] = None*) → Optional[List[ai_flow.metadata.workflow_snapshot

Retrieves the list of snapshots of the workflow.

Parameters

- **workflow_name** – The workflow to be listed.
- **namespace** – The namespace which contains the workflow.
- **limit** – The maximum records to be listed.
- **offset** – The offset to start to list.

Returns The WorkflowSnapshotMeta list, return None if no workflow snapshots found.

ai_flow.ops.workflow_trigger_ops module

ai_flow.ops.workflow_trigger_ops.**add_workflow_trigger**(*rule: ai_flow.model.rule.WorkflowRule*, *workflow_name: str*, *namespace: str = 'default'*) → ai_flow.metadata.workflow_event_trigger.WorkflowEv

Creates a new workflow event trigger in metadata.

Parameters

- **rule** – The rule that used to to judge whether start a new workflow execution
- **workflow_name** – The name of the workflow to be registered trigger.
- **namespace** – The namespace of the workflow.

Returns The WorkflowEventTriggerMeta instance just added.

ai_flow.ops.workflow_trigger_ops.**delete_workflow_trigger**(*trigger_id*)

Deletes the workflow trigger from metadata.

Parameters **trigger_id** – The id of the workflow trigger.

Raises AIFlowException if failed to delete the workflow trigger.

ai_flow.ops.workflow_trigger_ops.**delete_workflow_triggers**(*workflow_name: str*, *namespace: str = 'default'*)

Deletes all event triggers of the workflow.

Parameters

- **workflow_name** – The name of the workflow.
- **namespace** – The namespace which contains the workflow.

Raises AIFlowException if failed to delete workflow triggers.

ai_flow.ops.workflow_trigger_ops.**get_workflow_trigger**(*trigger_id: int*) → Optional[ai_flow.metadata.workflow_event_trigger.Workf

Retrieves the workflow trigger from metadata.

Parameters **trigger_id** – The id of the trigger.

Returns The WorkflowEventTriggerMeta instance, return None if no trigger found.

ai_flow.ops.workflow_trigger_ops.**list_workflow_triggers**(*workflow_name: str*, *namespace: str = 'default'*, *limit: Optional[int] = None*, *offset: Optional[int] = None*) → Optional[List[ai_flow.metadata.workflow_event_trigge

Retrieves the list of triggers of the workflow.

Parameters

- **workflow_name** – The workflow to be listed triggers.
- **namespace** – The namespace which contains the workflow.
- **limit** – The maximum records to be listed.
- **offset** – The offset to start to list.

Returns The WorkflowEventTriggerMeta list, return None if no workflow trigger found.

ai_flow.ops.workflow_trigger_ops.**pause_workflow_trigger**(*trigger_id: int*)

Pauses the workflow trigger.

Parameters **trigger_id** – The id of the workflow trigger.

Raises AIFlowException if failed to pause the workflow trigger.

ai_flow.ops.workflow_trigger_ops.**resume_workflow_trigger**(*trigger_id: int*)

Resumes the workflow trigger which is paused before.

Parameters **trigger_id** – The id of the workflow trigger.

Raises AIFlowException if failed to resume the workflow trigger.

Submodules

ai_flow.settings module

ai_flow.version module

The ai_flow version follows the PEP440. .. seealso:: https://www.python.org/dev/peps/pep-0440

CHAPTER

TEN

EXTRA PACKAGES

Here's the list of all the extra dependencies of AIFlow.

10.1 Database Extras

Those are extras that are needed when using specific database as backend.

extra	install command	description
mysql	pip install 'ai-flow-nightly[mysql]'	MySQL as metadata backend
mongo	pip install 'ai-flow-nightly[mongo]'	MongoDB as metadata backend

10.2 Blob Extras

Those are extras that are needed when using specific blob managers.

extra	install command	description
hdfs	pip install 'ai-flow-nightly[hdfs]'	HDFS as blob manager
oss	pip install 'ai-flow-nightly[oss]'	OSS as blob manager
s3	pip install 'ai-flow-nightly[s3]'	S3 as blob manager

10.3 Job Plugin Extras

Those are extras that add dependencies needed for integration with specific job plugins.

extra	install command	description
flink	pip install 'ai-flow-nightly[flink]'	Flink job plugin

10.4 Scheduler Extras

Those are extras for scheduler(only apache-airflow for now).

extra	install command	description
celery	pip install 'ai-flow-nightly[celery]'	Celery as the executor of apache-airflow

10.5 Bundle Extras

Those are extras that install one ore more extras as a bundle.

extra	install command	description
example_requires	pip install 'ai-flow-nightly[example_requires]'	Should be installed when running provided AIFlow examples
devel	pip install 'ai-flow-nightly[devel]'	Minimum development dependencies, including flake8, pytest, coverage, etc.
test	pip install 'ai-flow-nightly[test]'	Should be installed when you are running unittests of AIFlow
docker	pip install 'ai-flow-nightly[docker]'	Dependencies for docker compose

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

E

F

G

I

L

M

O

P

Q

R

S

T

U

V

W